I-Structure Software Caches: Exploiting Global Data Locality in Non-Blocking Multithreaded Architectures

Wen-Yen Lin

CENG 00-02

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4484)

May 2000

I-STRUCTURE SOFTWARE CACHES: EXPLOITING GLOBAL DATA LOCALITY IN NON-BLOCKING MULTITHREADED ARCHITECTURES

by

Wen-Yen Lin

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER ENGINEERING)

MAY 2000

Copyright 2000 Wen-Yen Lin

I-Structure Software Caches: Exploiting Global Data Locality in Non-Blocking Multithreaded Architectures

Wen-Yen Lin

CENG 00-02

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4484)

May 2000

I-STRUCTURE SOFTWARE CACHES: EXPLOITING GLOBAL DATA LOCALITY IN NON-BLOCKING MULTITHREADED ARCHITECTURES

by

Wen-Yen Lin

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER ENGINEERING)

MAY 2000

Copyright 2000 Wen-Yen Lin

Dedication

To my daughter Erin and my lovely wife Shu-Chiao.

Acknowledgements

My advisor, Professor Jean-Luc Gaudiot, was the inspiration for this thesis. I would like to thank for his inspiration, guidance, and support. It was a privilege to have been one of his students. I would also like to express my gratitude to Professor Massoud Pedram and Ming-Deh Huang for serving on my dissertation committee. Their inquisitive questions have stimulated my research. I also thank Professor Timothy Pinkston, Viktor K. Prasanna, and Rafael H. Saavedra for being on my Ph.D guidance committee.

The CAPSL (Computer Architecture and Parallel Systems Laboratory) at the University of Delaware led by Prof. Guang R. Gao generously provided me with the access to EARTH platforms. Their continuous efforts on developing, implementing, and maintaining the EARTH machines provided me a reliable experimental environment which leads to significant results presented in this dissertation. I thank Dr. José Nelson Amaral for valued discussion and reviewing portions of my analysis.

I would like to thank my colleague and very good friend of mine, Chung-Ta Cheng, for many insightful conversations and the friendship. I also would like to thank my former group members Dr. Yung-Syau Chen, Dr. Hung-Yu Tseng, Dr. Dae-Kyun Yoon, Dr. Halima Elnaga and Dr. Hiecheol Kim for their advisement and encouragement. I thank Dr. NamHoon Yoo for his well-developed simulator. Without his solid work, I would had spent more time on developing my own simulator. I also acknowledge PDPC group members Chulho Shin, James Burns, and Steve Jenks. Special thanks goes to Mary Zittercob and Joanna Wingert for their assistance.

My sincere gratitude goes to my parents, my brother and my sister for their love and long time support. I must thank my wife Shu-Chiao Huang for her understanding and love, which I can never thank enough. Without her continuous support, I would never have been able to finish the program. Finally, I would like to thank my daughter Erin, who brought me joyful moment and strength during my last stage of study.

Contents

Dedication ii

Acknowledgements iii

List Of Figures vii

List Of Tables ix

Abstract x

1 Introduction **1**
- 1.1 Synopsis 4

2 Background Research **5**
- 2.1 Multithreaded architectures 5
  - 2.1.1 Blocking Multithreaded Architectures 7
  - 2.1.2 Non-blocking Multithreaded Architectures 9
- 2.2 I-Structure memory system 12
- 2.3 Motivation 13
- 2.4 Related Work 15
  - 2.4.1 On Memory Models and Cache Management for Shared Memory Multiprocessors 16
  - 2.4.2 IS-Cache Design on the ETS System 17
  - 2.4.3 Scalable I-Structure Cache design 18
  - 2.4.4 A Cache Design for Input Token Synchronizations 19
  - 2.4.5 Empirical Study of a Dataflow Language on the CM-5 20

3 I-Structure Software Caches (ISSC) **21**
- 3.1 I-Structure Cache Design 21
  - 3.1.1 Deferred Requests Handling 22
  - 3.1.2 Deferred Queue Storage 23
  - 3.1.3 Deferred Read Sharing Problem 24
  - 3.1.4 Legality of Write Operations 25

3.2 The I-Structure Software Cache (ISSC) Runtime System . . . 26
3.2.1 Write-direct Policy . . . 26
3.2.2 Set-Associative Cache Allocation . . . 28
3.2.3 Cache Advance . . . 29
3.2.4 Deferred Read Sharing . . . 31
3.2.5 "Centralized" Deferred Requests and Distributed Deferred Reads . . . 32
3.2.6 Virtual Addressing . . . 33
3.2.7 Cache Replacement Policy . . . 33
3.2.8 ISSC System Overview . . . 35
3.3 Simulation Results . . . 36
3.3.1 The Simulator . . . 36
3.3.2 Simulation result . . . 36
3.3.2.1 The data locality . . . 38
3.3.2.2 The network traffic . . . 40
3.3.2.3 The system performance . . . 43
3.3.3 The effect of cache advance . . . 44
3.3.4 Cache Replacement . . . 45
3.4 Summary . . . 48

4 ISSC implementation on EARTH systems **50**
4.1 EARTH Architecture . . . 50
4.1.1 Fine Grain Multi-Threading . . . 50
4.1.2 Split Phase Communication and Synchronization . . . 52
4.2 Single Assignment Storage Structures . . . 54
4.3 ISSC Implementation on EARTH . . . 57
4.3.1 ISSC implementation using Threaded-C language . . . 57
4.3.2 Usage of ISSC in Threaded-C language . . . 60

5 Experiment results on EARTH systems **63**
5.1 Highlights of Experimental Results . . . 63
5.2 The Cost of ISSC Operations . . . 64
5.3 Description of Benchmarks . . . 67
5.4 Robustness to Latency Variation . . . 69
5.5 Summary . . . 74

6 Performance Modeling **75**
6.1 Performance Analysis . . . 76
6.2 The Analytical Models . . . 78
6.2.1 Verifying the Model . . . 80
6.2.2 Performance Predictions . . . 82
6.3 Summary . . . 85

7 Conclusions and future research **86**
7.1 Conclusions 86
7.2 Future research 88

Reference List **92**

Appendix A
ISSC's Implementation on EARTH using Threaded-C Language 103
A.1 ISSC Structure 104
A.2 ISSC Operations 108

Appendix B
Using ISSC with Hopfield Benchmark 112
B.1 Hopfield Benchmark 113
B.2 Makefile 127

List Of Figures

2.1 Distributed Deferred Queue Storage 18

3.1 Centralized Deferred Queue Storage 23
3.2 Data Block Integration 25
3.3 Structure of I-Structure Software Caches 28
3.4 Cache Advance Allocation 30
3.5 Deferred Read Sharing 31
3.6 The overview of I-Structure Software Cache runtime system 34
3.7 The Hit Ratio of Remote Requests: (a) Matrix Multiplication, (b) Conjugate Gradient, (c) 1-D FFT and (d) LU-Decomposition 39
3.8 The Number of Network Packets: (a) Matrix Multiplication, (b) Conjugate Gradient, (c) 1-D FFT and (d) LU-Decomposition 41
3.9 Speed up measurements: (a) Matrix Multiplication, (b) Conjugate Gradient, (c) 1-D FFT and (d) LU-Decomposition 43
3.10 The Effect of Cache Advance: (a) Matrix Multiplication and (b) Conjugate Gradient 44
3.11 Cache Replacement and Hit Ratio in MM Benchmark with Varying Cache size. 46
3.12 Cache Replacement and Hit Ratio in CG Benchmark with Varying Cache size. 46

4.1 The EARTH Model 52
4.2 (a) (1) An active fiber in the EU of P_i requests an EARTH split-phase block-move-sync operation; (2) The SU of P_i decodes the source address to the memory of P_j and sends a request for the block; (3) The SU of P_j receives the request and reads the block from the local memory. (b) (4) The SU of P_j sends the block over the network to the SU of P_i; (5) The SU of P_i writes the block in the local memory; (6) The SU of P_i decrements a synchronization slot counter, that becomes zero and causes the spawning of a fiber that will use the block transferred. 53
4.3 State Transition Diagram for the I-Structure Implementation 55
4.4 State Transition Diagram for the I-Structure Software Cache 58
4.5 Threaded-C with ISSC program example 61

5.1 Speedup in the MANNA machine. 70
5.2 Absolute speedup with 10 μs communication interface overhead . . . 71
5.3 Execution time with synthetically variable communication interface overhead 73

6.1 Execution time with add-on synthetically variable communication interface overhead. (a)Dense Matrix Multiplication (b)Conjugate Gradient (c)Hopfield (d)Sparse Matrix Multiplication 77
6.2 Performance prediction for different benchmarks 83
6.3 Performance prediction for communication optimization 83
6.4 Performance prediction for technology improvement 84

List Of Tables

2.1 Comparison of Blocking and Non-blocking Multithreaded Executions 12

5.1 Latency of EARTH and ISSC operations on EARTH-MANNA-SPN, measured in number of cycles (1 cycle = 20 ns). 65
5.2 I-Structure Software Cache Hit Ratios (%) 69
5.3 Average number of remote memory requests per node 69

6.1 Timing equations and the cross-points (μs) 78
6.2 Benchmark-related Parameters 80
6.3 Platform-related Parameters Measured from MANNA machine 80

Abstract

Non-Blocking Multithreaded execution models have been proposed as an effective means to overlap computation and communication in distributed memory systems without any hardware support. Split-phase operations are used to enable the tolerance of request latencies by a decoupling between the initiators and the receivers of communication/synchronization transactions. However, the data locality of the shared distributed global data is not exploited by conventional caches; moreover, each request also incurs the cost of communication interface overhead.

In this dissertation, we design our ISSC (I-Structure Software Cache) system to further reduce communication overhead for non-blocking multithreaded execution and develop a simulator to validates our design. The single assignment property of I-Structure eliminates the needs for cache coherence protocol and greatly reduces the overhead of this software cache. It is this property that make the concept of software cache feasible. This software cache combines the benefits of latency reduction and latency tolerance in non-blocking multithreaded system without any hardware support.

We then implement our ISSC on top of EARTH systems, which is a fine-grain multithreading system that could be implemented from off-the-shelf microprocessors, and we studied the performance of ISSC on EARTH-MANNA machine with a set of benchmarks. Our studies indicate that ISSC improves the system performance and makes the system more robust. We further develop analytical models for the performance of a multithreading system with and without ISSC. We compare our model's

prediction with our experimental results on EARTH-MANNA machines. These analytical models allow us to predict at what ratio of communication latency/processing speed the implementation of ISSC becomes profitable for applications with different characteristics. As a consequence the system can be ported to a wider range of machine platforms and deliver speedup for both regular and irregular application.

Chapter 1

Introduction

Multithreaded architectures have been proposed as a means to overlap computation and communication in distributed memory systems. By switching to the execution of other ready threads, the communication latency can be hidden from useful computations as long as there is enough parallelism in an application.

Non-blocking multithreaded execution models, like TAM [22], P-RISC [58], and EARTH [37], had been proposed to support multithreaded execution in a conventional RISC-based multiprocessor system without the needs of any specific hardware support for fast context switching. In these models, remote memory requests are structured as split-phased transactions so that the processor could continue executing other instructions which do not depend upon the request in progress. The request carries a tag, *continuation vector*, indicating the return address of the requested data in the consumer thread. The arrival of the requested data will be sent directly to the consumer thread identified by the continuation vector. The arrival of the last requested data in that consuming thread will then activate the thread and the thread will be ready to be executed. Therefore, in these models, thread activations are data driven: A thread is activated only when all the data elements it needs are available locally. Indeed, once a thread starts to execute, it executes to the end. In such a *non-blocking multithreaded* execution model, once the execution of a thread terminates, no thread contexts need to be saved before switching to the

execution of another active thread. Therefore, multithreaded execution is achieved without the needs of any specific hardware support.

A good execution model must be based on a good memory system to achieve high system performance [25]. An *I-Structure memory system* [9] provides split-phase memory accesses to tolerate communication latency. It also provides non-strict data access, which allows each element of a data structure to be accessed once the data element is available without waiting for the whole data structure to be produced. Each element of an I-Structure has a presence bit associated with it to indicate the state of an element, such as *Empty* and *Present.* Data can only be written into empty elements, and the slots are set to the *present* state after the data has been written into them. Read from an empty element is deferred until the data is produced. The split-phase accesses to the I-Structure elements provide fully asynchronous operations on the I-Structure. The non-strict data access on the I-Structure provides the system with a better chance to exploit fine-grain parallelism. The fully asynchronous operations on the I-Structures make it easier to write a parallel program without worrying about data synchronizations since the data are still synchronized in the I-Structure itself. The single assignment rule of the I-Structure provides a side-effect free memory environment and maintains the determinacy of the programs. All of these features make I-Structures, along with non-blocking multithreading, an ideal model for parallel computing.

While the combination of non-blocking multithreaded execution and I-Structure memory system appears to be a very attractive architecture for high performance computing, the major drawback of this system is that locality of remote data is not utilized. Since all remote requests are translated into split-phased transactions which are different from local memory read/write operations, the accesses of remote data do not pass through the local cache system and every remote data access is actually sent to the remote host. On the other hand, in some other multithreaded architectures, like ALEWIFE [2], FLASH [50], and *T-N.G. [8], every memory access

is issued as a local memory operation. Thread switching occurs when the processor stalls on cache misses or synchronization failures at run-time. This kind of models is what we call "*Blocking multithreaded* ." Thread switching involves context saving of the suspended thread, context loading of the next thread, and pipeline flushing. The overhead is much larger than the non-blocking multithreaded execution and it usually needs specific hardware support for fast context switching. Fortunately, the use of a local cache system exploits the global data locality and hence reduces the number of remote requests as well as the number of context switches.

With the very small overhead of context switching in the non-blocking multithreaded models, the highest overhead of these models is in the communication interface. The sending and receiving of network packets may take from dozens to thousands of cycles depending on the design of the network interface [21]. Since all requests are actually sent to the remote hosts through the network, all the sending and receiving requests incur the network interface overhead. Moreover, the requests for I-Structure memory accesses on the network also congest network traffic which may ultimately degrade system performance.

The goal of this research proposal is to develop an efficient cache scheme for the I-Structure memory system in the non-blocking multithreaded multiprocessor systems so that it could exploit the global data locality, reduce the number of network packets, and hence improve the overall system performance. The target environment we have in mind is a message-passing distributed memory multiprocessor system. The non-blocking multithreaded execution model is a compiler controlled multithreaded execution, and it could be implemented on any conventional RISC-based multiprocessor system without any add-on hardware support for multithreaded execution. We would intend to include our cache system to this model without any specific hardware support for further improvement of system performance. Therefore, a software implementation of this cache scheme, the *I-Structure Software Cache (ISSC)*, is proposed here to exploit global data locality without adding any specific

hardware support in the non-blocking multithreaded execution model. However, we do not limit this cache scheme to a software implementation only. This proposed research also provides a fundamental study for the hardware implementation.

1.1 Synopsis

This dissertation is orgnized as follows:

Before we start to discuss the design of I-Structure caches, we did a broad background research and report some replated work in Chapter 2. From these background research, we disscuss some design issues of I-Structure cache design in Chapter 3. We also describe the approaches adopted in our design, and provide details of our I-Structure Software Cache implementation. We then perform simulation of our ISSC design with selected benchmarks to validate our design.

We are not just satisfied with our simulation results, we also look for a real system that we could implement our ISSC on it. In Chapter 4, we have a brief introduction to our target system, EARTH, and described the implementation of our ISSC on the system using Threaded-C language. We measure the costs of some ISSC operations in Chapter 5 on EARTH-MANNA machine to have a better understanding of the overhead of software cache. In this chapter, we also test our ISSC with a set of real benchmarks and measure its performance. We show our ISSC make the system more robust to the latency variation.

In Chapter 6, we develop analytical models for a multithreading system with and without ISSC. We verify these models with the experimental results we measured in chapter 5 and make performance predictions for different benchmark characteristics and a wider range of machine platforms. Then we conclude this dissertation with the contributions of this research work and provide some directions for future research.

the communication latency behind the computation and reduce the idle time of processors.

Multithreaded processors should be attractive as nodes for a massively parallel machine programmed using the SPMD model. The *single-program multiple-data* (SPMD) model [24] is currently gaining wide acceptance for massively parallel scientific computation. The model is implemented by mapping it onto and MIMD multiprocessor, either manually or by using a dta parallel compilation strategy [36, 75, 45, 48, 26]. The SPMD model provides a good target language for the array computation features of such languages as Fortran 90 [73], High Performance Fortran [35], and certain functional programming languages (Sisal [56, 17, 18, 19] and Id [59], for example).

In some multithreaded models, like TAM [22], P-RISC [58], and EARTH [37], all remote memory accesses are translated into split-phased transactions at compilation time and a thread will be activated only when all the data inputs are available locally. Therefore, once a thread starts to execute, it executes to the end. This kind of execution model is called "Non-Blocking Multithreaded". *I-Structure memory system* is a split-phased accessing memory system. It provides non-strict data accesses, fully asynchronous memory operations, and fine-grain parallelism. This makes I-Structure memory, along with non-blocking multithreading, an ideal model for parallel computing. On the other hand, in some other multithreaded architectures, like ALEWIFE [2], FLASH [50], and *T-N.G. [8], every memory access is issued as a local memory operation. Thread switching occurs when the processor stalls on cache misses or synchronization failures at run-time. This kind of models is what we call "*Blocking multithreaded* ." With dedicated hardware support in this model, context switch overhead could be minimized to tens of machine cycles. Therefore, the communication latency could be overlapped by the interleaved executions of several threads.

2.1.1 Blocking Multithreaded Architectures

By "Blocking", we mean "Blocking Multithreaded Architectures" where the executions of threads are suspendable and also resumable. The idea is that during the execution of a thread, if the processor stalls on waiting for remote requests, synchronization failures, or even the local data to be brought from local memory to cache on cache misses, the processor would rather suspend the execution of the current thread and switch to the execution of other threads than just sit idle and wait for the action to complete.

However, to resume the execution of the suspended thread, the thread context needs to be saved while it was suspending. Since when the thread will be suspended cannot easily be predicted at compilation time, all the registers, status words, and some memory space have to be saved when a thread suspension occurs. Moreover, the thread context which is chosen for execution next has to be loaded into processor after the context of the suspended thread was saved. All of the jobs during thread switching (context saving and loading) should be done very efficiently, otherwise the processor may want to stick on the same thread and just be idle while waiting for the remote requests, synchronization failures, or cache misses to be finished.

Therefore, most of the architectures supporting blocking multithreaded execution, like Horizon [49], Tera [4]. MASA [33], J-Machine [61], ALEWIFE [2], FLASH [50], and *T-N.G. [8], have dedicated hardware which supports fast context switch. For example, in ALEWIFE, a modified SPARC processor, Sparcle [3] processor is used for supporting blocking multithreaded execution. In Sparcle, the register set is divided into several frames that are conventionally used as register windows [42],[65] for speeding up procedure calls in SPARC. In their design, they partition the register file into four hardware contexts. A context switch to a precess whose state is currently stored in one of the register frames on the processor is effected in a small number of cycles. Each Sparcle processor will support up to four hardware threads and unlimited virtual processes. The mapping of process contexts to register frames

is managed by software. By this dedicated hardware design, the context switching could be achieved within 14 cycles. However, this kind of hardware support would increase the complexity and the cost of processor design.

However, there are still some side effects of the blocking multithreading which are hard to minimize by dedicated hardwares. One of the side effects is pipeline flushing. In pipelined processors, all the instructions entering pipeline become invalid right after the thread was suspended and the first instruction of next thread has to be fetched into the first stage of pipeline. This will result in bubbles in the pipeline. The deeper the pipeline, the higher the overhead suffered by the system. The other side effect is cache contention [1]. In the blocking multithreaded execution, all the existing threads (including the active thread, ready threads, and suspended threads) of the processor compete for the limited cache space with each other. This gives rise to a higher cache miss rate.

Fortunately, the exploitation of the global data locality reduces the number of remote requests and the number of context switches. In the blocking multithreaded execution, all the remote memory accesses are treated as local accesses. In machines with caches, the actual remote requests are sent to the remote hosts only when they are missed from the local cache. The block of data located in remote hosts will be brought back to the local cache with the requested data and the following remote memory accesses may hit the local cache. Therefore the thread execution could be continued without suspension. The reduction of the actual remote requests also gives a lower network traffic rate. However, the use of caches in the multiprocessor systems raises another important issue in multiprocessor system design, namely the *cache coherence* problem [28, 20].

In conclusion, by maintaining multiple process contexts in processors supporting blocking multithreaded execution with fast context switch, the thread execution will be suspended when it stalls on the remote requests, cache misses, or synchronization failure, and the processor switches to the execution of other threads. Such that the

communication latency could be overlapped with useful computation and processor utilization increases. Since there are dedicated hardwares to support the context switching at the run-time, all of the memory accesses could be treated as local memory accesses and the actual remote requests are made only when they are missed from the local cache. Therefore, the global data locality could be easily exploited by the local cache and hence reduces the chance of thread switching and network traffic. However, provided sufficient parallelism exists, the number of threads in the processor is still limited by the complexity of the processor and the increased cache miss rate [1]. Indeed, with the requirement of dedicated hardware support for fast context switching and maintained cache coherence, it would take a long time and would be very costly to build this kind of systems.

2.1.2 Non-blocking Multithreaded Architectures

Split-phased transaction [34, 72] is an asynchronous memory access scheme in message passing multiprocessor systems. In the systems, remote memory requests are structured as split-phased transactions so that multiple requests may be in progress at one time. An instruction issues a request to the processor or memory module containing the desired data, and then other instructions which do not depend upon the result of the request in progress are executed. The request carries a tag, *continuation vector*, indicating the return address in the consuming thread at which the computation should be continued when the response arrives. By splitting the remote memory request into two phases, requesting and consuming, the processor could continue executing other useful computations without wait for the data to arrive while the request is in progress. The arrival of the requested data will be sent directly to the consuming thread identified by the continuation vector. This feature of the split-phased transaction provides the ability for overlapping the communication latency with useful computations.

Basically, the non-blocking multithreaded execution model was evolved from the concept of data-flow execution model. Data-flow execution can be thought of as a very fine-grain multithreaded execution model. Indeed, in data-flow models, each thread contains only one instruction. The instruction will be activated only when the operands it needs are generated. After the execution of the instructions finished, the output data token is passed to other instructions and activates them. This would result in a *sequence of activation*. To improve the performance of the data-flow architecture, most processors designed for data-flow execution are pipelined, like MONSOON [63, 64, 34] and RAPID [60]. The stages of the pipelined processor are interleaved with different sequences of activations. Therefore, the high through put could be achieved. However, due to the high cost of the matching unit for the operands matching of instructions and the poor performance of the single sequence of activation execution, researchers from this area proposed the non-blocking multithreaded execution model. Examples of multithreaded architectures based on dataflow models are: Iannucci's work [39, 40] in combining dataflow ideas with sequential thread execution to define a hybrid computation model, the EM-4 project [47, 46, 68] at the Electrotechnical Laboratory (ETL) in Japan, the successor of MONSOON project, *T [14], TAM [22], P-RISC [58], and EARTH [37].

The main idea of non-blocking multithreaded execution is to group the sequence of activation without any remote memory accesses, branches, and synchronization into one thread at compilation time. So that, once a thread starts to execute, it executes to the end. A thread is like an atomic execution unit, which is like an instruction in the data-flow execution model. Since the thread execution will not be suspended, no context needs to be saved for the thread at the run-time. As for the beginning of a thread execution, since the execution is not resumed from previous execution, no process context needs to be loaded. Therefore, there is almost no overhead during the thread switching. This is the reason why it is easier to implement this kind of model from off-the-shelf processors [55]. Moreover,

since the thread boundary has been determined at compilation time, in the pipelined processors, the first instruction of the thread which will be executed right after the current thread could be pre-fetched into to the pipeline while the last instruction of the current thread is in the execution stage. So that, the pipeline is also highly utilized while thread switching without any bubble stages.

For the remote memory accesses, the requesting and consuming of the remote data are broken into different threads by using the split-phased transaction. In a split-phased transaction memory access, along with the requested data address, the continuation vector (the return address of the requested data) was sent to the remote host by the requesting thread. According to the continuation vector, the requested data are sent back directly to the consumer thread by the remote host. And, the consumer thread may become active if all the data it needs are available locally. Since the requesting threads are only responsible for sending out the requests, it is not necessary for them to wait for the requests to complete after sending out the requests. The processor could continue the execution of current thread or other active threads while those split-phased transactions are in process. Therefore, the communication latency could be hidden from the execution of other threads.

However, the major drawback of this non-blocking multithreaded execution model is that the global data locality is not exploited. Since every remote memory access has been compiled into split-phased transaction explicitly, each remote access actually send out the request to the remote host and the remote host sends back the reply message along with the requested data. These requests are different from the local memory read/write operations, and, therefore, these remote memory accesses do not pass through the local cache systems and the local cache system takes no advantages of the remote data locality. On the other hand, in the blocking multithreaded model, every memory access is issued as local memory operation. The request is sent to the remote host only when the data is not in the local cache. The

local cache system exploits not only the data locality from local memory but also the global data locality.

Characteristics	Blocking Multithreading	Non-Blocking Multithreading
Thread execution	Interleaved (suspendable)	Atomic entity
Thread switching	Run-time controlled	Compilation time controlled
Hardware support context switch	Needed	Unnecessary
Remote memory accesses	Local accesses	Split-phased transactions
Pipeline flushing	Yes	No
Granularity	Coarse	Fine
Global data locality exploitation	Easy	Difficult
Network traffic	Medium	High

Table 2.1: Comparison of Blocking and Non-blocking Multithreaded Executions

Finally, I would like to sum up the comparison between the blocking multi-threaded model and the non-blocking multithreaded model in table 2.1.

2.2 I-Structure memory system

An I-Structure memory system [11, 9, 34] is a conventional data structure with some constrains on its construction and destruction. It is designed for the data storage of scientific applications in parallel computing to achieve efficient accesses, provide fine-grain parallelism, and preserve the determinacy of computing. I-Structure memory system explicitly use split-phased transaction for the memory access, and this provides the system with the ability for hiding the latency of accessing I-Structure memory from useful computation.

Each element of the I-Structure has a presence bit associated with it to indicate the state of the element, such as *Empty* and *Present*. There are three primitives for the operations in I-Structure memory system.

- *I-allocation* allocates consecutive data elements for an array structure and these data elements are initialized as in *Empty* state.
- *I-store* stores a produced data item into the empty data element. After the data item is written into the data element, its state is set to *Present* state.

If an *I-store* attempts to write data into a data element which is already in the *Present* state, it causes an error. This constrain makes the I-Structure memory as a single assignment memory system. Each data element could be accessed after the data is stored into it without waiting for the data of the whole structure to be produced.

- *I-fetch* reads the data from the data element in I-Structure memory. If the I-fetch is issued to an *Empty* data element, the request is *deferred* until that element has been written. A deferred request can be memorized simply by saving its *continuation vector* in the data element. Once the value is present, it can be sent to the requester using the saved continuation vector. This would allow a request being issued before the data is produced.

The non-strict data access on the I-Structure provides the system with a better chance to exploit fine-grain parallelism. The fully asynchronous operations on the I-Structure make it easier to write a parallel program without worrying about data synchronization since the data are still synchronized in the I-Structure itself. The single assignment rule of the I-Structure provides a side-effect free memory environment and maintains the determinacy of the programs. Giving enough parallelism in the program, the split-phased nature of memory requests allows us to hide the extra latency of remote memory accesses in the distributed multiprocessor environment. All of these features make I-Structures highly suited to distributed memory systems designed to exploit fine-grain parallelism, like the non-blocking multithreaded execution.

2.3 Motivation

It appears that using I-Structure memory system along with non-blocking multithreaded execution becomes a promising architecture for high performance parallel computing. This architecture exploits fine-grain parallelism, hides the extra latency

of remote requests from useful computation, increases programmability while maintaining the determinacy of the parallel applications, and of course, is low cost to build.

However, the major drawback of this architecture is that the remote data locality is not utilized. We need to exploit the global data locality in this architecture for several reasons. Indeed, with the capability for latency tolerance brought by split-phased transactions and the small overhead of thread switching in the non-blocking multithreaded execution, it would appear that the length of the remote request is irrelevant. However, it turns out that communication latency tolerance is based on one central assumption: it could be hidden from computation *as long as* there are enough ready threads. When there is not enough parallelism, *single thread performance* is closely related to the communication latency and it becomes critical. Exploiting the global data locality will reduce the mean time between thread activation and therefore the processor utilization increases in the critical section. Secondly, even with enough threads to tolerate communication latencies and low thread switching overhead, the highest overhead of this architecture is in the communication interface. The sending and receiving of network packets may take from dozens to thousands of cycles depending on the design of the network interface [21]. Since all requests are actually sent to the remote hosts through the network, all the sending and receiving requests incur the network interface overhead. Finally, even though many machines include dedicated hardware to handle the network communication so that communication interface overhead is taken away from the computation processors, the requests for all the remote data accesses on the network also congest network traffic, which may ultimately degrade system performance.

Therefore, an I-Structure cache system which caches these split-phase transactions in non-blocking multithreaded execution is required to further reduce communication latencies and release the network traffic. This cache system would provide ability for communication latency reduction while maintaining the communication

latency tolerance ability in this architecture. Therefore, our goal is to develop a novel I-Structure cache scheme to exploit global data locality in non-blocking multithreaded architectures. The target environment we have in mind is a message-passing distributed memory multiprocessor system. The non-blocking multithreaded execution model is a compiler controlled multithreaded execution, and it could be implemented on any conventional RISC-based multiprocessor system without any add-on hardware support for multithreaded execution. The single assignment property of the I-Structure eliminates the cache coherence problem from the cache design. This would make it possible to implement the cache system as a software run-time system without being detrimental to the system performance, and we would intend to include our cache system to this model without any specific hardware support for further improvement of system performance.

Therefore, in this proposed research, we developed an I-Structure Software Cache (ISSC) [51] in the non-blocking multithreaded execution model with I-Structure-like memory environment without adding any hardware. We would like to see the impact of the ISSC on the overall system performance by analyzing the data locality utilization, network traffic, overhead distribution, and speed-up curves of some applications.

2.4 Related Work

There is some research about the I-Structure Cache design which has been pursued elsewhere, but non of the designs are intended to implement as a run-time system.

2.4.1 On Memory Models and Cache Management for Shared Memory Multiprocessors

Dennis and Gao [25] proposed a cache management scheme for their Abstract Shared-Memory computer system, which is a dataflow program execution model and specified the I-Structure model as the memory system to support the synchronizing memory operations. They proposed a high-level concept of the I-Structure Cache management scheme but without detail implementation.

In their design, a cache line will be allocated first in the local cache when a read miss occurs. The continuation vector of the original request will be stored in this allocated cache line, and a new request will be forwarded to the remote host by using the address of the allocated cache line as the continuation vector but not the original one. The later requests for the same data item will be deferred in the cache line while the first request is in progress. After the first request is replied from the remote host, the data item is written into the pre-allocated cache line and is also forwarded to all the continuation vectors which have been deferred in that cache line. A write-through with write allocate policy is adopted in their design in a write miss situation. In the I-Structures, the deferred requests from all other hosts for the same data element are queued in the host or memory module which owns that data element.

In their design, the size of a cache line is a single I-Structure element. Therefore, only the temporal data locality is exploited and the spatial locality is not touched. All other details of the cache design, like cache organization and cache replacement algorithm, are not mentioned in their design. And also, no simulation or evaluation are performed in their work.

2.4.2 IS-Cache Design on the ETS System

Kavi et al. [43] proposed a design of cache memories for multithreaded dataflow architectures. The design includes an I-Structure cache memory to exploit the data locality of the shared data structures in multiprocessor environment. Basically, the design of their I-Structure cache (IS-Cache) is a hardware supported cache system using the Explicit Token Store (ETS) model of dataflow systems.

The IS-Cache keeps not only the I-Structure elements requested (I-fetch operations) by the processor but also the I-Structure elements produced (I-store operations) by the processor. A write-back on demand policy is adopted for the I-store operations. The data items produced by local host are kept at the local IS-Cache and are written back to the I-Structure only when there are requests for those data items or they are replaced from the IS-Cache. As in conventional cache system design, a cache line is allocated only when the data are brought back from remote host. Therefore, in a read miss situation, the request is forwarded to the I-Structure directly without doing anything on the local IS-Cache. If the requested data item has been produced and is available in the I-Structure, the data item is sent back to the consumer thread and a copy of the data item is also kept at IS-Cache. If the requested data item is not yet available (the data element is in *Empty* state) in the I-Structure, the request is deferred in the I-Structure and a message is sent to the producer of that data item to indicate that there is deferred request of that data item in the I-Structure. If the data item is already in the producer's IS-Cache, that data item is written back to the I-Structure and the deferred request for that data element are fulfilled. Otherwise, a missing table is maintained in the producer's IS-Cache to indicate the pending status of the I-Structure elements. After the data item is produced and stored in the producer's IS-Cache, the missing table is checked and the data item is written back to the I-Structure and the deferred request could be fulfilled.

To implement the write-back on demand in the IS-Cache, extra space for the missing table is needed. Also, in order to send the pending status of a data item to its producer, an additional directory for the information of the producers of the I-Structure elements is required. This would further make it difficult to implement the dynamic allocation of data structures in the I-Structure. It would also be difficult to implement a thread migration strategy which will change the producers of the data items at run-time. Moreover, addition interrogation messages will be introduced to the network when requests for empty I-Structure elements occur.

2.4.3 Scalable I-Structure Cache design

Papadopoulos [62] and Cheng [31] independently proposed scalable methods to deal with the storage of the deferred requests in the I-Structure.

In the multiprocessor systems, multiple hosts may issue requests for the same data item in the I-Structure. If that data item is not produced yet, all the requests have to be deferred in the I-Structure. As the number of pending requests grows, there may be not enough space to store all the pending requests in the I-Structure which owns that data item. Moreover, when the data item is produced and written to the I-Structure finally, all the deferred requests will be served. This may cause a hot spot problem on the network.

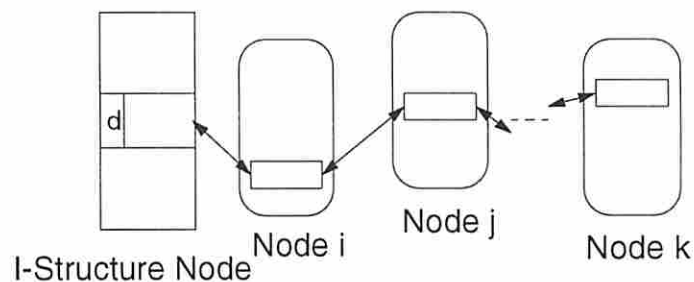


Figure 2.1: Distributed Deferred Queue Storage

Therefore, they proposed distributed mechanisms for the storage of the pending requests: they are distributed among the requesting hosts. As shown in figure 2.1, every requesting host provides one (or more) slot to store each one of its own pending request(s), and all of these requests are linked in a queue. This scheme make the growing of the deferred queue quite scalable, since for every requesting host, only one slot is needed for the queue. It also avoids hot spots in the network if there are too many requests pending in a single data location.

2.4.4 A Cache Design for Input Token Synchronizations

Roh and Najjar's project [67] on the design of storage hierarchy in multithreaded architectures was trying to exploit the locality of the frame storage on the Pebbles multithreaded model. The Pebble multithreaded model is a non-blocking multithreaded model which is the same as the architecture that we have in mind. However, the locality exploited in their work is the frame storage which is used to store the input tokens of the threads. This reduces the match time of each incoming token. They showed the execution time becomes linearly proportional to the match time when the match time is greater than 3 cycles. In their simulation, the average match time could be reduced to 1 cycle based on the design of a fully associative cache. In this work, the locality of the global shared data is not touched. We believe that the execution time is dominated by the match time when the match cycle is large as shown in their work. We think that the execution time with a small match cycle is dominated by the availability of the threads. The I-Structure cache exploits the global data locality and hence reduces the average turn around time of the remote requests. The smaller the remote request turn-around time, the less threads are needed to overlap the communication latencies. Therefore, by incorporating the I-Structure cache with their work, the system could be further improved.

2.4.5 Empirical Study of a Dataflow Language on the CM-5

Culler et al. [23] implemented the idea of I-Structure caching in software manner on *Id90* compiler for their Threaded Abstract Machine (TAM) implemented on the CM-5. The idea of I-Structure caching is similar to our work but they also did the single I-Structure data element caching which is the same as Dennis and Gao's work as we introduced in previous section. In their implementation, the unit of a cache block is a single I-Structure data element. Therefore, only temporal data locality had been exploited. With a cache block size of one I-Structure data element, no deferred read sharing problem will occur. This made their design comparatively easier, like cache replacement, deferred read handling, etc. However, from our simulation, it shows that spatial data locality does play an important role in the performance improvement. Moreover, temporal data locality could be easily utilized by the programmer or the compiler without implementing the I-Structure caching, as we shown in our FFT benchmark.

Chapter 3

I-Structure Software Caches (ISSC)

3.1 I-Structure Cache Design

In one aspect, I-Structure cache design is simpler than the cache design for conventional memory systems. That is, no cache coherence problem is encountered in the I-Structure cache design. This is because of the *inherent cache coherence* feature of I-Structure Cache. Indeed, I-Structure is a single assignment memory system. In single assignment memory systems, multiple updates of a data element are not permitted. Once a data element is defined in a single assignment memory system, it will never be updated again. The copies of the data elements in the local cache will never be updated. Therefore, cache coherence is already embedded in I-Structure memory systems. It makes the design of I-Structure cache much simpler without having to take care of the cache coherence problem.

However, in other aspects, the design of the I-Structure Cache is not as straightforward as the cache design for conventional memory systems. This is because of some characteristics of I-Structure, such as split-phased transaction, single assignment property, deferred read, and the presence bits of data elements. Therefore some design concerns and issues will arise in the I-Structure cache design.

3.1.1 Deferred Requests Handling

In an I-Structure, a request may be *deferred* in the I-Structure if the request arrives while an data item has not yet been written into the I-Structure. The deferred request will be satisfied after the data item is produced and written back to the I-Structure. The services of deferred reads have to be guaranteed, otherwise some threads may wait for the already produced data elements forever and this may result in some deadlock situations.

In the I-Structure memory system without I-Structure cache, there will be not problem at all for this guaranty since the I-store operations will write the produced data back to the I-Structure directly and all the pending requests in the I-Structure will be fulfilled as soon as the data are written into the I-Structure. However, adding the I-Structure cache to the system may keep the data of I-store operations in local cache without writing them back to the I-Structure immediately. In the case that no cache replacement occurs, the produced data might be kept at local cache and would not be written back to the I-Structure forever. If it happens to have some pending requests for those data in the I-Structure, then these pending requests will never be satisfied. Therefore, the design of I-Structure cache has to avoid this situation carefully.

One of the solutions is the write-back on demand policy as used in Kevi's IS-Cache design [43]. The produced data which are kept in the cache by local host will be written back to the I-Structure not only when they are replaced from the cache, but also when there are requests for these data from other hosts. After the data are written back to the I-Structure, the deferred reads could be satisfied. This scheme will prevent the unnecessary data being written back to the I-Structure if there will be no requests from other hosts. However, a write-through cache design provides a simple solution to guarantee the service, because the produced data element will be written to the I-Structure as soon as it is produced. Once the data element is

written to the I-Structure, the deferred reads queued on the data element slot can be satisfied.

These two solutions provide the guaranty of the deferred request services. The write-back on demand cache design will reduce some unnecessary network traffic, but it is more complex and expensive to implement than the write-through cache design is. In section 4, we will have more discussion on this issue and explain the reasons of why we chose write-through policy in our design.

3.1.2 Deferred Queue Storage

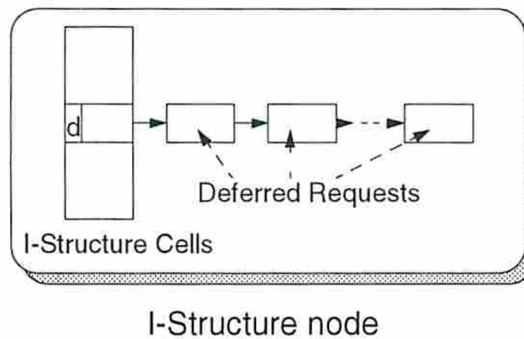


Figure 3.1: Centralized Deferred Queue Storage

In a multiprocessor system, there may be several requests pending on a data element before the data is generated. How the system maintains the queue of these deferred requests is also an issue. The conventional method is called the "*Centralized*" storage method: all of the deferred requests are stored in the owner's place, as shown in figure 3.1. This method is very simple, and since all the deferred requests are kept at the owner's place of the data element, all of the pending requests can be satisfied as soon as the data is written into its location. However, the number of pending requests depends solely on the application. Further, as the number of pending requests grows, there may be not enough space to store all these requests. Therefore, this scheme may not be scalable: even though there is enough space to

store all the pending requests, whenever the data is generated, all of the pending requests on this data have to be serviced simultaneously. This may cause a hot spot problem on the network.

The "*Distributed*" storage method independently proposed by Papadopoulos [62] and Cheng [31] provides a scalable solution for the unlimited growing of pending requests and also avoids the hot spot problem on the network caused by the services of those pending requests. Moreover, since the deferred queue is distributed among the requesting processors, the I-Structure needs only serve the first pending request which is stored in the I-Structure data element. After the reply of the first pending request arrives the requesting host, the pending request, which is from other host and stored in that host, could be satisfied. This makes the services of the pending requests on different data elements as in pipeline fashion, and therefore, it increases the throughput of the I-Structure memory operations. However, as in Cheng's design, the storage slots of this distributed deferred queue are provided by the I-Structure cache of each requesting host. The cache lines allocated for the distributed deferred queue may be replaced, and the queue will be broken. So that, additional effort must be expended to recover the queue once it is broken. Moreover, those requests which are pending at the end of the queue may wait for a long time for the requests to be served.

However, the chance of the pending requests to explode the space in the "Centralized" storage method will play an important role in the decision of using the "Distributed" storage method or not.

3.1.3 Deferred Read Sharing Problem

In I-Structure memory systems, every data element has a presence bit associated with it to indicate its state (*Present*, *Empty*, or *Deferred*). Indeed, to exploit the spatial data locality, a whole block of data elements should be requested by the cache instead of the requested data item only. As shown in figure 3.2, the data elements in

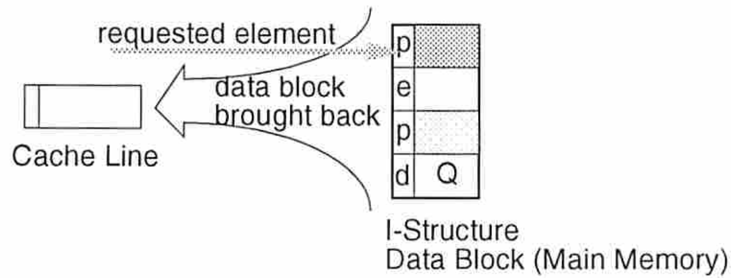


Figure 3.2: Data Block Integration

the same block may be in a different state; some of them may be in the *present* state and some of them may be in the *empty* state. How the data elements in the different states will be integrated into a whole data block needs to be careful handled in the I-Structure cache design.

Deferred read sharing is one of the issue happens in the integration of data elements in different states into one data block. Without doubt, the *present* data should be brought back into the cache and a deferred request is stored in the slot of the requested data element if it is in the *empty* state. The issue comes when there are other empty data elements within the same data block. Is the deferred read going to be put on every empty data element, or just put on the requested data elements and the other empty elements left still empty? This would be up to the choice of the designer and would have different impact on the cache performance.

3.1.4 Legality of Write Operations

As we discussed before, an I-Structure memory system is a single-assignment memory environment in and of itself. For instance, it must be ensured that write operations are only made to empty locations. If this can be guaranteed by the compiler or the language, then the write operations could be delayed in the local cache until the data is needed by other hosts or the cache block is replaced. However, if the legality of write operations is not ensured by the compiler/language, a write-back cache design may result in some non-deterministic behavior. In this case, a write to

an already defined element may occur, but the doubly-written data may be kept in the local cache forever while the rest of the system is not aware of this situation. A write-through cache will be much safer in an I-Structure cache system if the legality of write operations was not enforced by the compiler or the language.

3.2 The I-Structure Software Cache (ISSC) Runtime System

The I-Structure Software Cache runtime system proposed here will take advantage of the spatial and temporal localities of the global data in the I-Structure memory systems, without any hardware support. The runtime system works as an interface between the user applications and the network interface. A block of memory space is reserved by this run-time system as the software cache space. It filters every remote request and reserves a memory space in local memory as a cache of remote data. A remote memory request is sent out to the remote host only if the requested data is not available on the software cache of the local host. Instead of asking for the requested data item only, the whole data block surrounding the requested data item is brought back to the local host and stored in the software cache. Therefore, spatial data locality can also be exploited.

There are several features of our ISSC system that I want to discuss first before I give an example to explain the overview of whole ISSC system.

3.2.1 Write-direct Policy

As described in section 3 regarding the *deferred request handling* and the *legality of write operation* issues in the I-Structure cache design, different write policies would have different impacts on the I-Structure cache design. In our I-Structure Software Cache design, we didn't adopt the *write-back* policy for the following reasons:

- I-Structures are a producer-consumer type of memory system. There may be some requests pending on an element before the data item is produced. We want to satisfy those pending requests as soon as the data becomes available. If a write-back policy was adopted in our software cache, some of the threads may still be waiting for the data even though they have been produced. This may result in a shortage of ready threads in some processors, thereby rendering them idle. Even though a *write-back on demand* may solve this problem, in order to write the data back to the I-Structure as soon as there are requests from other hosts, the producers of data elements have to be known before the data are actually produced and extra tables needs to be checked in write operations and read miss situations. This would increase the overhead of the cache system, and of course, this is not what we want for a software cache run-time system. Knowing the producers of every data element in advance also makes it difficult to dynamically allocate space for the data structures and also makes it difficult to migrate threads in the run-time, which may change the producers of data during run-time.

- The main reason for using a write-back cache is to prevent the unnecessary memory updates which happen when the data in the local cache are updated again before they are read by other processors. However, in the single assignment memory system, the data in the cache will never be updated. Therefore, the write-back cache design does not have an advantage over a write-through cache design in the single assignment memory system.

- We want to ensure that write operations are made only into empty locations as soon as the write operation has been issued. If the write operation is cached in the local software cache and written back to the remote host only when the cache block is replaced or there are requests from other hosts, the write operation might attempt to modify an element which is already in the *present*

state due to some error. This may result in using this illegal data from the local cache by the local host.

For these reasons, a *write-direct* or *write-through* policy could be adopted in our I-Structure Software Cache system. Therefore, data will be written to the I-Structures as soon as the write operations are issued. This simply guarantees the services of deferred reads after the requested data elements are produced. However, we simply use *write-direct* policy instead of *write-through* to prevent the node that issued the write from replying to a read request before the legality of te write is verified. In other words, there is no caching for write operations. This simple write-direct policy ensures writing to empty location only, satisfies deferred reads as soon as possible, and avoids deadlock situations.

3.2.2 Set-Associative Cache Allocation

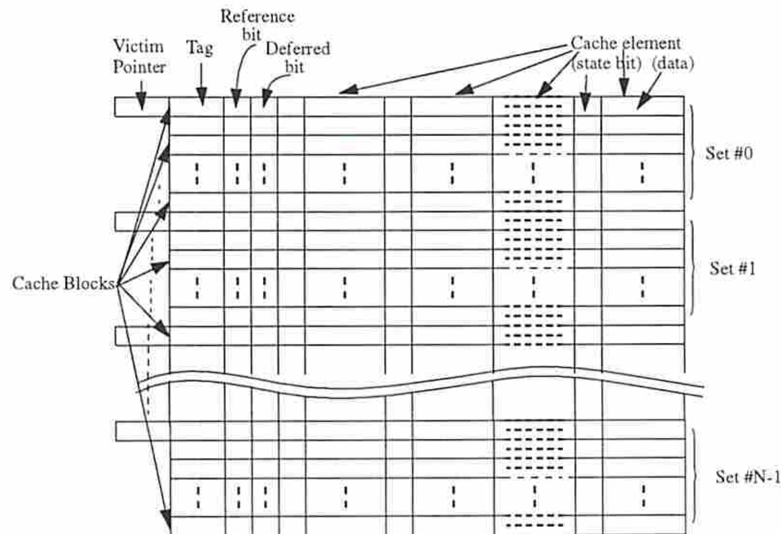


Figure 3.3: Structure of I-Structure Software Caches

Cache search schemes play a very important role in the cache performance. In hardware design, a fully associative cache has the highest performance because of its parallel search and the full utilization of the cache space, but it is very expensive to

implement. In a software implementation, a parallel search is obviously impossible inside a single processor. The direct-mapping scheme has the fastest search time in software implementation, however, it has the worst cache utilization. In order to have a higher search performance and better cache utilization, a set-associative search mechanism is adopted in ISSC.

A requested data address is mapped to a set of cache blocks by a hash function. If the address matches with the tag of one of the cache blocks in the set, then we have a cache hit. Otherwise, it is a cache miss, and a cache line is allocated for this request as described in the *cache advance* feature. Figure 3.3 shows the structure of the ISSC. Each cache line has a deferred-bit, a reference-bit and a tag field which indicates the address of the first element in the block, and it contains block of cache elements. Cache lines are allocated in pre-reserved consecutive memory blocks to store the data of cache blocks so that they could be directly accessed by index addressing. Each set has a victim pointer which is used in cache replacement.

3.2.3 Cache Advance

In conventional cache designs, the cache space is allocated when the data block is brought back to the local host. However, the cache space is allocated in advance in the ISSC when a read miss is detected. This is what we call the "Cache Advance." Indeed, due to the long latency and unpredictable characteristics of the network in a distributed memory system, a second remote access to the data elements in the same data block may be issued while the first request is still traveling through the network. In conventional cache allocation methods, multiple outstanding memory requests for the same data block from the same host are possible. By using our approach, the second and later requests are deferred in the pre-allocated cache space while waiting for the data block to come back.

In the example shown in figure 3.4, a cache block size of 4 data elements is assumed. As in I-Structure memory, each data element in the cache is also associated

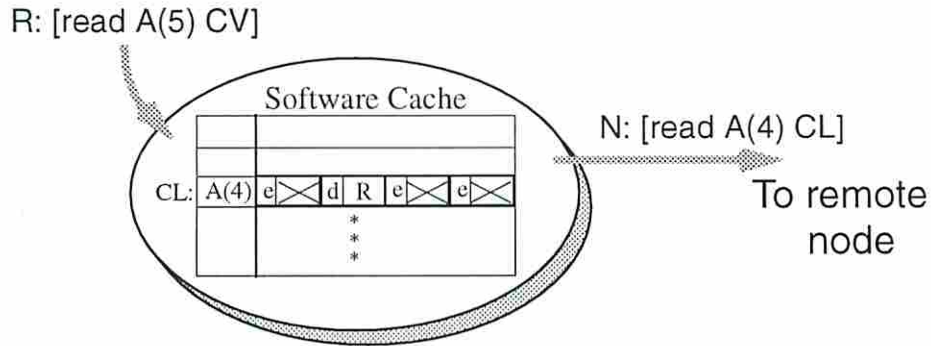


Figure 3.4: Cache Advance Allocation

with a presence bit. So that each element could be distinguished in different states: *Present*, *Empty*, and *Deferred*. This presence bit would provide a second-level data synchronization point for the data, so that the feature of fully asynchronous memory operation of the I-Structure memory could be maintained. To exploit the spacial data locality, a cache block is allocated in a read miss instead of one data element is allocated, and all data elements in this cache block are initialized in the *Empty* state. In this example, a read request "R" asking for the data "A(5)" is made and missed in the software cache. A cache block "CL" is allocated for this missing read before the request is sent to the remote host. Instead of sending the original request "R" to the remote host, a new request "N" asking for the data block beginning with "A(4)" along with the new continuation vector "CL" is sent to the remote host and the original request "R" is deferred in the second element of the pre-allocated cache block "CL". Therefore, the following requests asking for A(4), A(5), A(6), and A(7) will hit the cache and will be deferred in CL while the request "N" is in progress. This allows duplicate remote memory requests to be eliminated and therefore ultimately improves overall network performance.

3.2.4 Deferred Read Sharing

As described earlier, there are two ways to deal with the deferred read sharing problem. One way is to append the request to all the data locations which are empty. The other one is to defer the request on the requested data location only and leave other empty locations in the block still empty.

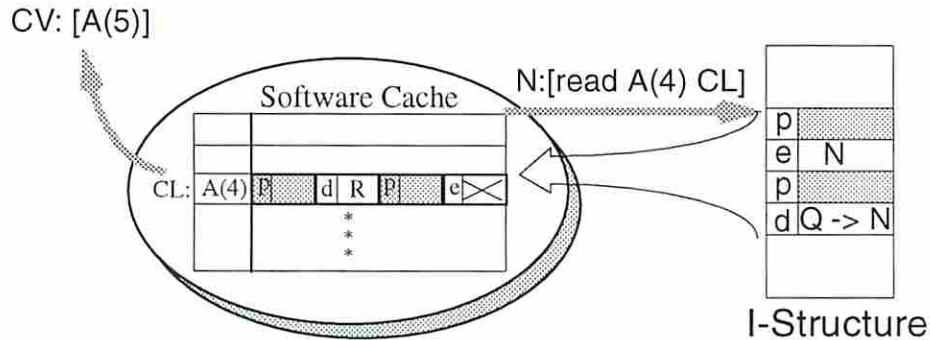


Figure 3.5: Deferred Read Sharing

In our ISSC, a deferred read is shared by all the empty data element located in the same data block. In the example shown in figure 3.5, a request "N" asking for the data block beginning with A(4) arrives in the I-Structure. Among the four data elements in this block, two of them, A(4) and A(6), are in *Present* state, one, A(5), is in *Empty* state, and the other one, A(7), is in *Deferred* state with a deferred request "Q". This request "N" is not only deferred in A(5), which was originally requested, but also A(7). And, the valid data of A(4) and A(6) are sent back to "CL" in the software cache of the requesting node. In the requesting node, read requests which hit the cache but find out that the data elements are in *Empty* or *Deferred* state would just be deferred in the local software cache without sending the requests to the I-Structure. Since the deferred read has been shared by all the empty data elements of a data block in the I-Structure, once the data elements are

filled with valid data, they will be sent back to the local software cache and the requests deferred in the local cache could be satisfied.

For applications with good spatial locality, placing a deferred read indication on all empty elements within the data block would yield better performance than would just putting the deferred read on the requested data element, since all the data within a data block only need one request. By appending the request to all of the empty locations, the data will be sent to caches after they are produced without making another request. However, for applications with poor spatial data locality, the deferred read of a whole block may introduce more network traffic because it may send to caches data which may never be needed by the local host. This is due to the fact that the data has not actually been requested but just happens to reside in the same data block alongside other requested data.

We believe that for most numerical applications, there is plenty of structural parallelism with spatial locality. Therefore, the deferred read sharing is implemented in our current ISSC runtime system. From our simulation results, we will demonstrate that the spatial data locality dominates the data locality in the matrix multiplication benchmark.

3.2.5 "Centralized" Deferred Requests and Distributed Deferred Reads

A simple "centralized" method is used for the implementation of the queues of deferred requests. Since ISSC is a software runtime system, the space to store the pending requests could be dynamically allocated if needed. There would be no scalability problem in our design. It should be noted that the implementation of this runtime system should be as thin as possible in order to reduce its overhead. However, to implement the distributed method, extra messages would be introduced into the network to link the requests together. A link recovery scheme is also needed for the distributed method when the link is broken. All of these would introduce

more overhead to the runtime system which is of course undesirable. Therefore, the centralized deferred read method is used in the ISSC. Indeed, with the "cache advance" and "deferred read sharing" features of ISSC, the length of the queue of deferred requests for each element in the I-Structure is bounded by the number of nodes in the system. This is because at most only one request is sent from each node to the host node. Future deferred reads are kept locally in the node.[1] However, the potential hot spot problem of the "centralized" deferred read method has to be further considered in the future.

3.2.6 Virtual Addressing

Even though we recognize that the single assignment rule of I-Structures simplifies the cache coherence problem, some of the cache coherence problems still occur when the I-Structure memory space is de-allocated and re-utilized. To totally avoid the cache coherence problem, a logical address, like the data structure ID, must be used. It is the job of compiler to make sure that no two data structures have identical IDs.

3.2.7 Cache Replacement Policy

Because of the single assignment feature, the intermediate data structures, which are storages neither for input data nor for final output data, will be sooner deallocated during the computation than the data structures in multiple updatable memory systems. These intermediate data structures will have a short life time in the cache. Therefore, page faults in the I-Structure cache would occur more frequently than in conventional memory caches. This means that cache replacement is very important in the I-Structure cache design.

A simple *Pseudo-LRU* policy is adopted as the replacement policy in our implementation of ISSC. A cache block that has any element in the *deferred read* state

[1] In the rare situation in which all the lines in the ISSC are irreplaceable, reads bypass the ISSC and are sent directly to the host node.

is irreplaceable and the *deferred bit* is set. A single *reference bit* is attached to each cache block as shown in Figure /refhash. The reference bit is set whenever there is a read to the block. A victim pointer is used to select a block to be replaced. When a cache replacement is needed, the block pointed by the victim pointer is tested to verify if it is replaceable by checking the deferred bit and if its reference bit is zero. If either condition is not satisfied, the reference bit of the block is reseted and the victim pointer is advanced. When a replaceable block with a zero reference bit is found, it is replaced and the victim pointer is advanced to the next block. The victim pointer is pointed to the first block after cache initialization. However, if all the blocks in the set are irreplaceable, reads bypass the ISSC and are sent directly to the host as if there is no I-Structure cache.

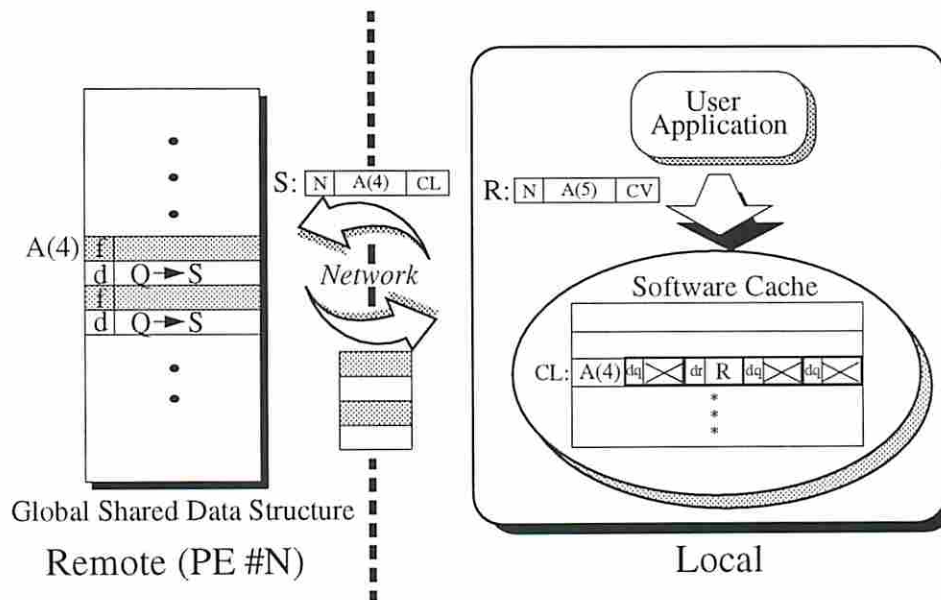


Figure 3.6: The overview of I-Structure Software Cache runtime system

3.2.8 ISSC System Overview

An overview of the operation of the ISSC is shown in figure 3.6. In this example an application makes a request R for the element $A(5)$ of the I-structure A located in the remote host N. Since a split-phase transaction is used, the request R must include the destination host N, the address of the requested data $A(5)$, and the continuation vector CV of the requested data. Without the ISSC, the request R would be sent directly to the remote host N through the network. However, the ISSC intercepts the request R before it is sent to the network. In this example, $A(5)$ was in the invalid state. Instead of sending the original request R, a new request S asking for a data block which includes the requested data $A(5)$ is generated. Before sending the new request, a cache line space CL in the software cache is reserved for the newly requested block. In our example, the location of the requested data $A(5)$ is in the second slot of the cache line which begins with $A(4)$. The original request R is stored in a dynamically allocated queue. A pointer to the head of the queue is stored in the cache location of $A(5)$, and the state of this location is marked as *deferred read* "dr." All other elements in this block are marked as *deferred request* "dq." Meanwhile, the new request S, which contains the destination host number N, the beginning address of data block $A(4)$, and the reserved cache line location CL for this request, travels to the remote host N through the network.

In our example, when the block request S is received by the remote host N, it finds two valid data elements, $A(4)$ and $A(6)$, two empty data $A(5)$ and $A(7)$, and one deferred read Q pending for $A(5)$ and $A(7)$ in this data block. The ISSC in host N then reads the valid data elements, $A(4)$ and $A(6)$, and defers the request S for $A(5)$ and $A(7)$. The two valid data elements $A(4)$ and $A(6)$ are sent back to the requesting host.

When the local host receives the elements $A(4)$ and $A(6)$, the ISSC fills the corresponding slots. If there are any pending requests on those data elements, the ISSC satisfies them by sending the requested data to the CVs as specified in the

pending requests. When the data element $A(5)$ is produced and written to its location in the remote host N, the deferred read S in the remote host N is serviced and a data packet carrying $A(5)$ is sent back to CL in the original requester. Upon receiving the data packet containing $A(5)$, the ISSC of the original requester places the data element $A(5)$ into its slot at CL and satisfies the request R sending the data to the CV specified in R.

3.3 Simulation Results

We have performed some simulation experiments to validate our ISSC scheme.

3.3.1 The Simulator

Our simulator for the ISSC is built on top of the Generic MultiThreaded machine (GMT) simulator [74] developed at the University of Southern California. The GMT simulator provides a generic platform of non-blocking multithreaded machine parameterizing various architectural details. The global heap memory is an I-Structure-like system for shared global data storage. There are two instructions for global structure access, AREAD (array read) and AWRITE (array write). The AREAD instructions to the remote hosts are cached by the ISSC runtime system. In order to test the effects of different cache block sizes, the cache block size is configurable in the simulator. However, the cache size remains the same with varying cache block size configurations. This means that when the cache block size increases, the total number of available cache lines decreases.

3.3.2 Simulation result

The goal of the simulation is to demonstrate the impact of our ISSC on the distributed memory multi-processor system environments which use split-phase memory transactions to tolerate the communication latencies. We want to demonstrate

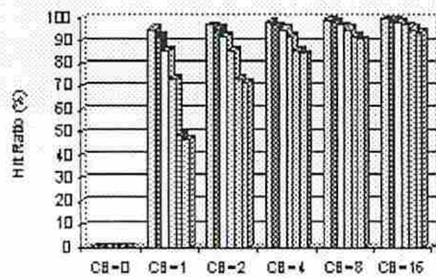
the kind of data locality which can be exploited by ISSC, and what kind of impact ISSC has on the network traffic. Further more, we want to verify the effect of ISSC on the system performance. Therefore, four benchmark programs with different characteristics were tested in the simulation. One is a matrix multiplication with a matrix size of 32x32 double precision floating-point numbers, and the other one is the kernel function of the conjugate gradient method for solving 256 linear equations with 256 unknown variables. The other two benchmarks were chosen from SPLASH-2 kernels, 1-D FFT with 512 complex data points, and LU-Decomposition for a 32x32 matrix. These four benchmarks have different categories of data reference locality. The matrix multiplication benchmark has excellent temporal locality of data reference, while the other three benchmarks have spatial data locality dominating the locality of data references. This is because that in matrix multiplication, two input matrices are constantly referenced during the whole computation, however, in the other benchmarks, intermediate vectors or matrices are generated and would not be referenced again after the computation pass by.

In our simulations, we wanted to test how the ISSC performed with varying cache block sizes in different system sizes. We want to simulate it on the ideal case by eliminating the performance degradation caused by a small cache size. Therefore, in the simulations, each PE is configured with 24K words of software cache which is large enough for the problem size we are testing. The communication latency between two PEs is set by the parameter "COM", which is the mean time of communication delay between two PEs. In this part, we chose a reasonable communication latency by setting "COM" to 2.0. With hardware configurations of 2, 4, 8, 16, 32, and 64 PEs, and different cache block sizes, 0 (no cache), 1, 2, 4, 8, and 16 words, the simulation results are shown in following figures.

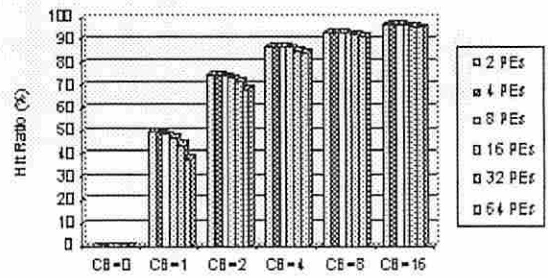
3.3.2.1 The data locality

Figure 3.7(a) shows the cache hit ratio of the remote requests of matrix multiplication for various hardware configurations. When no cache is configured (Block Size: CB=0), every remote request is sent to the remote host, so the hit ratios are obviously always 0%. For a cache block size of 1 (CB=1), only the temporal locality is exploited. For a small number of processors, like 2 PEs, the temporal locality dominates the whole data locality (this can be seen by comparing with the cache hit ratios of cache block size 1, 2, 4, 8, and 16 of figure 3.7(a)). However, in most MPP systems, there are tens, hundreds or even thousands of processors in a system. With the same problem size, the remote request hit ratio decreases linearly while the number of processors increases. This means that the spatial data locality becomes dominant. This is because the data are also distributed among the processors and, therefore, the number of remote requests increases. This shows that it is not enough to only rely on the exploitation of temporal data locality in MPP systems. Also, from the figure, we could see that the degradation of hit ratios becomes faster in smaller cache block size while the system is scaling up. With the help of our ISSC runtime system, a remote request hit ratio of 90% could be achieved on a cache block size 8 in a 64 PEs configuration and it also reduces the gap of the hit ratios between different number of processors.

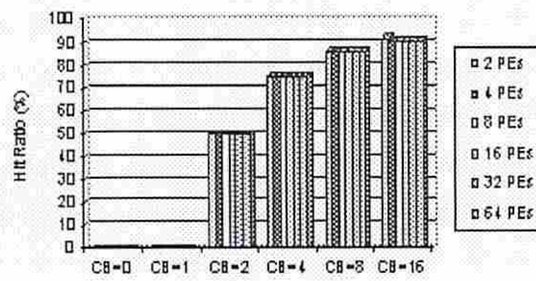
Figure 3.7(b) shows the cache hit ratio of the remote requests of the conjugate gradient benchmark. In the conjugate gradient method, most of the computation consists in updating array elements. With a good data partition scheme, the updates of array elements could be done locally. Therefore, when the number of processors increases, the number of remote requests does not increase much and the remote request hit ratio just decreases much more slowly compared to matrix multiplication in figure 3.7(a). The benchmark does not have so much temporal data locality of references as the matrix multiplication does. Therefore, the hit ratios are less than 50% when only temporal data locality was exploited. However, the increase in cache



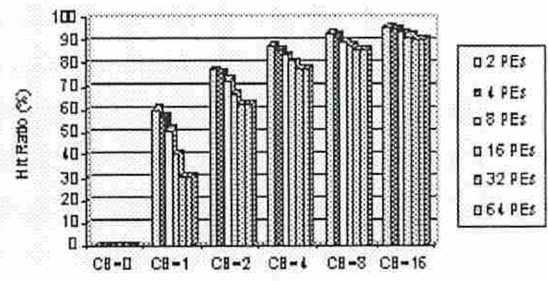
(a) Matrix Multiplication



(b) Conjugate Gradient



(c) 1-D FFT



(d) LU-Decomposition

Figure 3.7: The Hit Ratio of Remote Requests: (a) Matrix Multiplication, (b) Conjugate Gradient, (c) 1-D FFT and (d) LU-Decomposition

block size takes advantage of the increased spatial data locality and improves the overall hit ratio. Indeed, with a cache block size of 16, the hit ratio increase from 47% (with CB=1) to 97% in 8 PEs configuration.

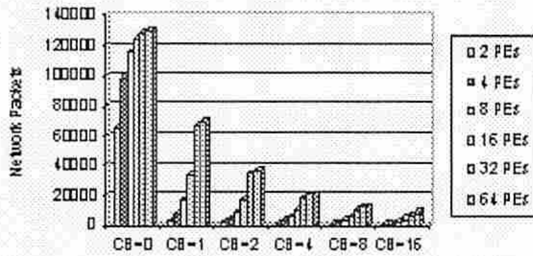
Figure 3.7(c) shows the cache hit ratio of the 1-D FFT benchmark. It shows that the hit ratio is still 0% when cache block size equals to 1. This is because that each data element is actually referenced twice during the entire computation and in our implementation, the data element has been stored as a local variable for the next reference after it was accessed from the remote host. This is a typical example of the fact that the temporal data locality could be utilized by the programmer or some compiler optimization techniques. It is interesting to note that the cache hit ratio remained the same while the number of processors scaled up. This is unlike other benchmarks when the cache hit ratio decreased as the number of processors increased. This is due to the way we distributed data among processors and the memory access patterns of 1-D FFT algorithm.

Finally, figure 3.7(d) shows the cache hit ratio of the LU-Decomposition benchmark. It is similar to the Conjugate Gradient benchmark. However, the cache hit ratios are almost the same for 32 and 64 PEs. This is because that the problem size we tested, 32x32, is small relative to the system size.

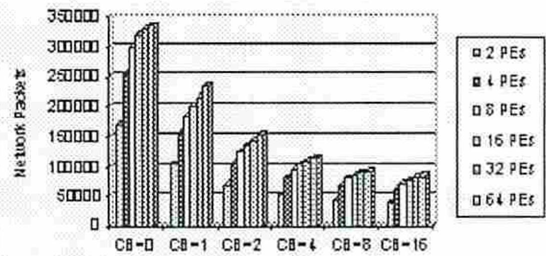
In summary, we could see that exploitation of spatial data locality is really necessary especially in larger system size. From our simulation results, over 90% hit ratios are achieved in all benchmarks with the cache block size of 16 words in all system configurations.

3.3.2.2 The network traffic

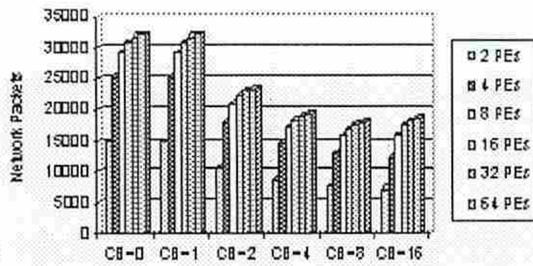
Agarwal [1] showed that the performance of multithreaded processors is traded off against network contention. In the non-blocking multithreaded execution model, the situation is even worse, because a finer granularity is being exploited and more communication is necessary between processors.



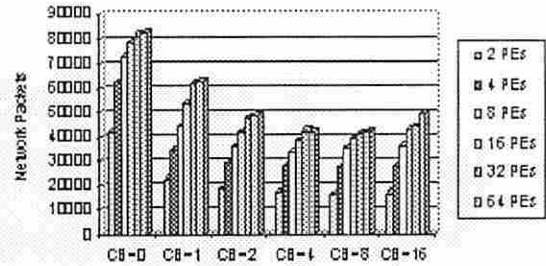
(a) Matrix Multiplication



(b) Conjugate Gradient



(c) 1-D FFT



(d) LU-Decomposition

Figure 3.8: The Number of Network Packets: (a) Matrix Multiplication, (b) Conjugate Gradient, (c) 1-D FFT and (d) LU-Decomposition

In figure 3.8(a), we show the number of network packets with the same problem size as the matrix multiplication benchmark program and the same hardware configurations as before. The number of network packets is counted at the network interfaces of each host. It is the total number of packets issued to the network by all the hosts. When no cache is configured, the total number of network packets increases while the number of processors increases. Increasing the number of processors means that we are trying to distribute the data and tasks among more processors in order to improve the system performance. This result matches the conclusion shown in Agarwal's analysis. However, by exploiting both the temporal and spatial global data locality, the number of network packets decreases dramatically. Our simulation results also show that the number of network packets for the 64 PEs system decreases from 130,032 without the ISSC runtime system to 9072 with the ISSC runtime system and a cache block size of 16. More than 90% of the network traffic is reduced by the ISSC runtime system.

Figure 3.8(b) shows the number of network packets in the conjugate gradient benchmark. Because of the fine grain parallelism of this benchmark, the ratio of the number of thread activation packets to the total number of network packets is larger than in the matrix multiplication benchmark. Therefore, the effect of network packet reduction is not as significant as in matrix multiplication. However, 70% of the network traffic is still reduced by the ISSC runtime system for the 64 PEs system with a cache block size of 16.

Figure 3.8(c) and (d) show the number of network packets in the 1-D FFT and LU-Decomposition benchmarks respectively. One interesting observation is that the number of network packets increases slightly in a 64 PEs system when the cache block size increases from 8 to 16 in both benchmarks. Increasing the cache block would only fetch into more data which will not be referenced. This will not harm the hit ratio that the system could achieve. However, because of the deferred read

sharing, those un-referenced data are still sent back to the cache after they are produced. This is what will increase the network traffic.

3.3.2.3 The system performance

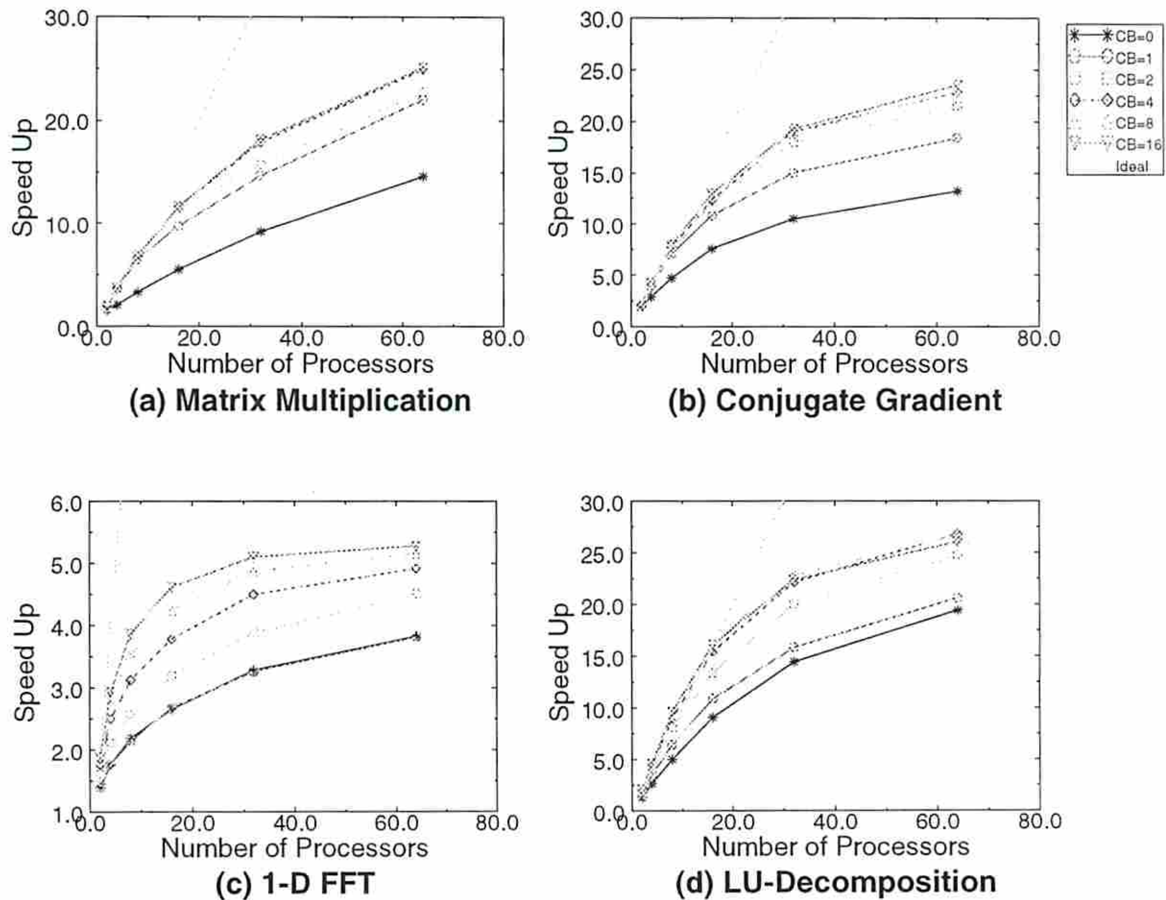


Figure 3.9: Speed up measurements: (a) Matrix Multiplication, (b) Conjugate Gradient, (c) 1-D FFT and (d) LU-Decomposition

Figure 3.9 shows the speed up measurements of our benchmarks. The speed up is measured by the execution time in different configurations related to the execution time in a single processor system without ISSC enabled. From our simulations, we could observed that our ISSC improved the system performance by a factor of 75% up to 95%. The utilization of data locality in the non-blocking multithreaded

execution shortens the mean time between two thread activations, and hence reduces the system idle time. Therefore, the total execution time was reduced by our ISSC. In figure 3.9, we could see that our ISSC could achieve optimal performance at cache block size of 8 words. Even though increasing the cache block size to 16 did yield a better cache hit ratio than on cache block size of 8, as we could see in figure 3.7, the improvement in system performance, however, is not that much. Indeed, in the LU-Decomposition benchmark, with a cache block size of 16 the system performance even degrades a little bit compared to a cache block size of 8, as shown in figure 3.9(d). This is because that, as shown in figure 3.8(d), the network traffic increases when the cache block size increases from 8 to 16, and therefore the system incurs more overhead by handling those extra data requests which may not be referenced eventually.

3.3.3 The effect of cache advance

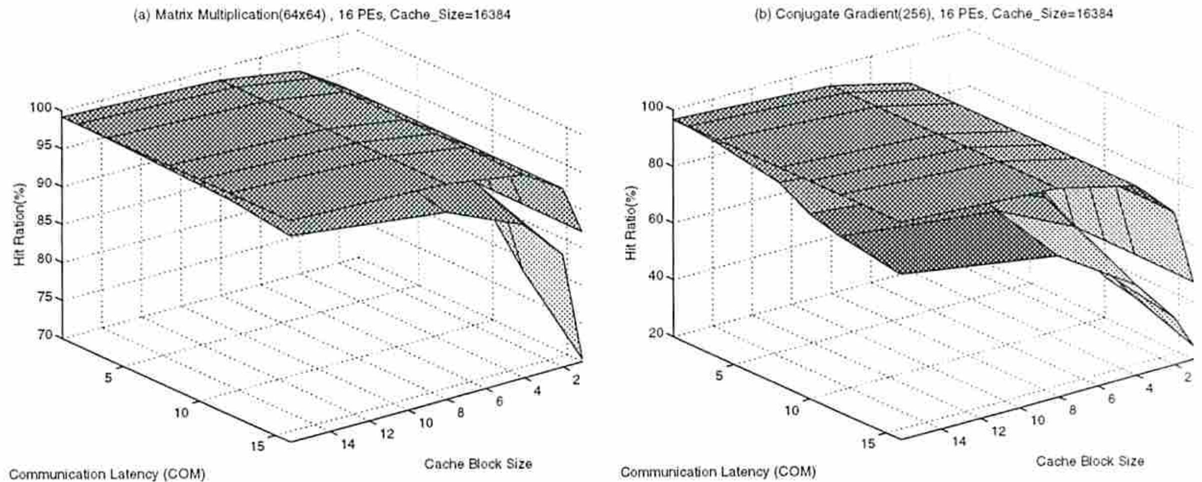


Figure 3.10: The Effect of Cache Advance: (a) Matrix Multiplication and (b) Conjugate Gradient

The cache advance feature in our design is a very unique feature in the I-Structure cache design. By allocating a cache block for a read miss before sending out the request to the remote host, the following requests for the data located in the same

block could be deferred in this pre-allocated cache block. To verify how this scheme affect the cache performance, we varied the communication latency by setting the "COM" parameter to different values (1.0, 2.0, 4.0, 8.0, 10.0, 12.0, and 16.0) in our simulator with the cache advance respectively enabled and disabled. The same benchmarks were simulated with variable cache block sizes in the system with 16 PEs and 16K words caches. The results are shown in figure 3.10. Again the hit ratios are plotted in a 3-D format, so that we could easily see how the hit ratios change with different configurations. The results with cache advance enabled and disabled are plotted in the same figure, so that we could easily compare the effect of cache advance. In figure 3.10 (a) and (b), the upper surfaces are the hit ratios with cache advance enabled and the lower surfaces are the hit ratios without cache advance. We can see that the cache hit ratios are not affected by the variation of communication latencies for a fixed cache block size with the cache advance turned on. However, without the cache advance, the cache hit ratio decreases while the communication latency becomes higher and higher.

3.3.4 Cache Replacement

In a real situation, the cache will not be sufficiently large to hold all the data referenced by a local host. Therefore, the cache replacement scheme plays a very important role in the cache design. In our design, a multiple-queue LRU algorithm is used. Cache lines are linked in different queues according to how many elements are in the empty state in the cache line. When a cache replacement occurs, the least recently used(LRU) cache line in the queue which keeps the cache lines with the most empty elements will be chosen as the victim. However, a cache line with a deferred read pending on any one of its elements will never be replaced to prevent deadlocks. Therefore, if all the cache lines have at least one deferred request inside,

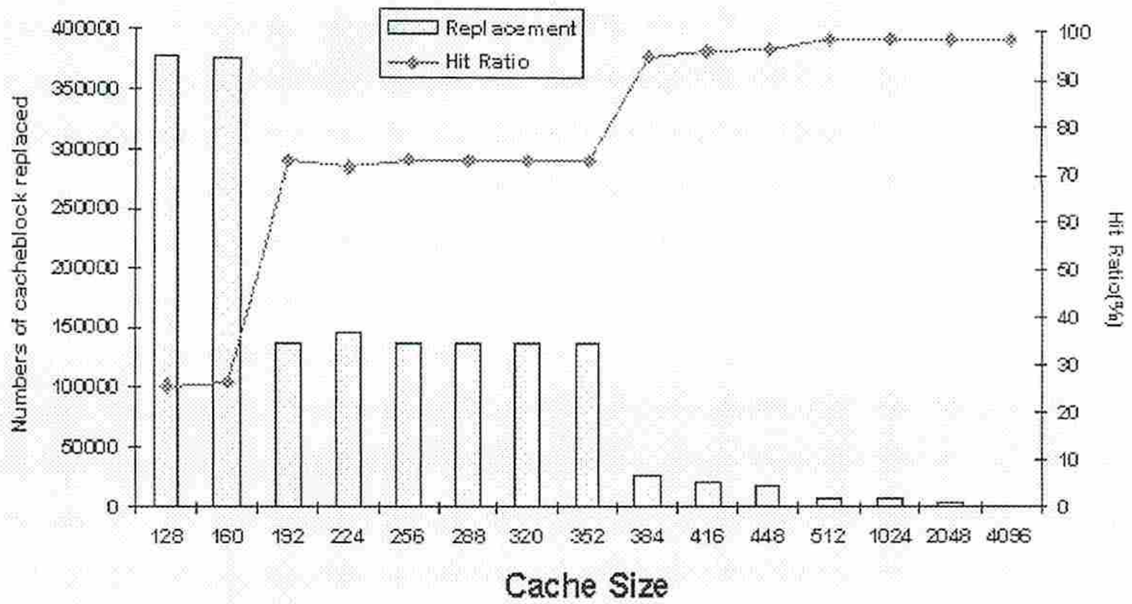


Figure 3.11: Cache Replacement and Hit Ratio in MM Benchmark with Varying Cache size.

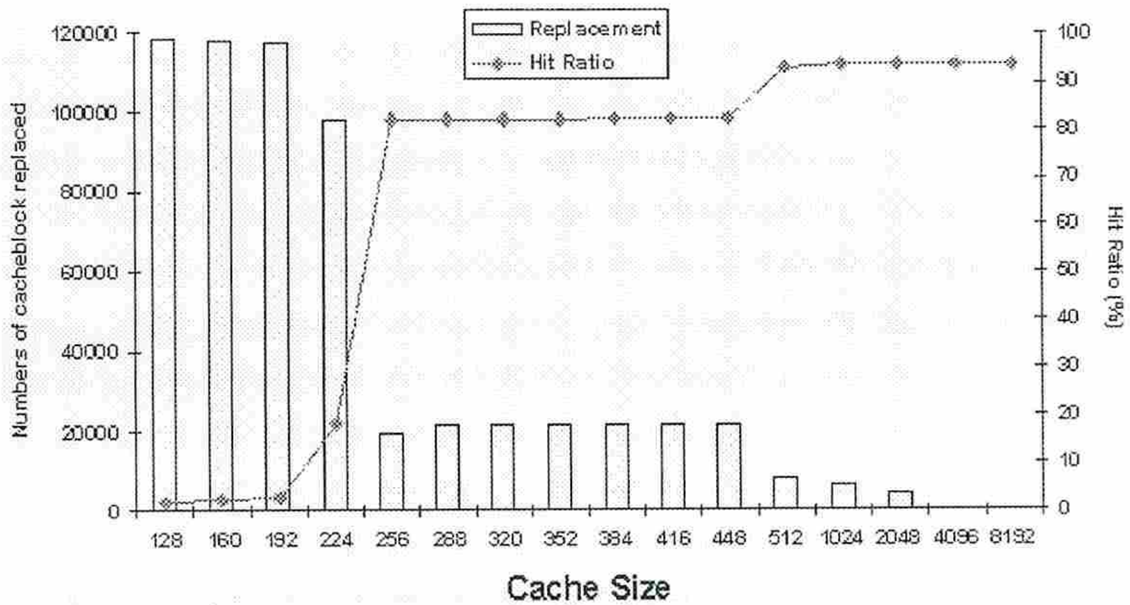


Figure 3.12: Cache Replacement and Hit Ratio in CG Benchmark with Varying Cache size.

the missed read will be directly forwarded to the remote host and no cache line is replaced.

In this series of simulations, we have fixed the system size at 16 PEs , the cache block size at 8 and set the communication latency parameter COM to 2.0. We varied the cache size from 128 words to 8192 words and recorded the number of cache blocks being replaced and the cache hit ratio for each configuration. The same benchmarks tested in the ideal case are simulated in this part with the same problem sizes. Figures 3.11 and 3.12 show the simulation results. The bar charts are the numbers of cache blocks being replaced and the lines are the cache hit ratios in different cache sizes. The hit ratios are very small when there are only small cache sizes available. With limited cache sizes, 70% and 80% of hit ratios achieved in matrix multiplication and conjugate gradient respectively. In figure 3.11, the hit ratio jumps from 27% to 72% when the cache size increases from 160 words to 192 words, and in figure 3.12, the hit ratio jumps from 19% to 81% when the cache size increases from 224 words to 256 words. Increasing the cache size a little, the data locality is fully exploited while there are still thousands of cache blocks being replaced. This shows that our ISSC still performs reasonably well with limited cache space by using multiple-queue LRU replacement scheme.

The results in figure 3.7, 3.8, 3.11, and 3.12 show that the ISSC not only helps the system by exploiting the data locality for split-phase type remote memory accesses in different type of applications, but that it also reduces the number of network packets in the network. In figure 3.10, we show the effect of the cache advance scheme on the system in the aspect of remote request hit ratios. By applying the cache advance scheme, we provides an adaptive cache system which will not be affected by the varying of communication latency. This is really useful in the MPP systems whose communication latency is usually long and unpredictable. How these advantages of ISSC would effect the overall system performance should be further examined and simulated.

3.4 Summary

In this chapter, we proposed a split-phase transaction caching scheme for the I-Structure-like memory systems. We discussed several issues of I-Structure cache design and described our design approaches. We also described the details of our ISSC implementation.

We validated our design by Generic MultiThreaded machine (GMT) simulator with several benchmarks. From the simulations, we have demonstrated the impact of our ISSC runtime system on the split-phased transaction memory accessing in the non-blocking multithreaded execution model. With a cache block size of 16, a hit ratio of 90% could be easily achieved in all benchmark programs. The number of network packages also decreases a lot comparing to the original quantity without ISSC. With all these effects, our ISSC increased the system utilization and improves the overall system performance up to 95%. The cache advance scheme in our ISSC also provides the adaptability to the unpredictable communication characteristics in DSM systems. This makes our ISSC achieve the same performance without being affected by the variation of the communication latency.

Although some of the simulation results are preliminary and need to be conducted with a wider array of benchmarks, we are encouraged by the dramatical reduction in network traffic, by the evidence of global data locality exploited by our ISSC and by the impact of our ISSC on the overall system performance.

We continues our studies by expanding the benchmarks to a variety of applications. In the meantime, the overhead of this software cache has to be further evaluated. However, as the speed of the processors increases dramatically, the gap between computation speed and the network overhead becomes larger and larger. The idea of this software cache becomes more promising. We looks for an appropreate platform to implement our ISSC and find the EARTH [37] as our target for the implementation. In the next chapter, we describe our implementation of ISSC

on the EARTH machines and in chapter 5 we show the performance measurement of our ISSC on EARTH-MANNA machines.

Chapter 4

ISSC implementation on EARTH systems

4.1 EARTH Architecture

The EARTH, *E*fficient *A*rchitecture for *R*unning *TH*reads, project [37, 69] lead by Prof. Guang Gao originally from McGill University, Canada in the Fall of 1993 and now continued at the University of Delaware is a fine-grain non-blocking multithreaded execution model for the efficient implementation of multithreading on off-the-shelf microprocessors with minimal additional hardware support for multithreading.

4.1.1 Fine Grain Multi-Threading

Modern multi-threaded systems can be classified into two broad classes according to the granularity of the threads that they can efficiently support while yielding good performance: coarse grain multi-threading and fine grain multi-threading. Typically in a coarse grain multi-threading system (1) the thread switching mechanism involves interactions with the operating system; and (2) there is a limited number of light-weighted processes to which threads must be bound. In a coarse grain multi-threading system, a thread can be viewed as a refinement of an operating system process. In contrast, in a fine grain multi-threading system: (1) the unit of computation is a collection of instructions grouped in a code block; (2) the system does

not impose limits on the number of threads that can be active at the same time; (3) the system does not require binding to any sort of limited resources; [1] and (4) the thread switching mechanism is quite efficient and does not involve the operating system, it typically requires that only a small amount of state information be saved in each switching. In a fine grain multi-threading system a thread can be viewed as the coarsening of an instruction.

The fine grain multi-threading system studied here, EARTH, is derived from the data-flow model of computation. In the classical strict data-flow model an instruction is enabled for execution when all its operands are available [30, 38, 29, 69]. To enforce this enabling condition, the instructions that produce operands must be able to send a synchronization signal to all the instructions that will consume their results. This model proved unwieldy for the implementation of machines based on current standard off-the-shelf hardware and compiler technology. In EARTH, the unit of computation is not an instruction, but a code-block formed by many instructions. An instantiation of the code-block running on a processing node is called a *fiber*, and multiple code-blocks are grouped into *threaded functions*. A successful program written in Threaded-C [70], the programming language for EARTH, will produce enough fibers to maintain the local processor busy while remote computations and data fetching operations are performed.

Figure 4.1 shows the EARTH model and it assumes that each processing node has an Execution Unit (EU) that executes the fibers and a Synchronization Unit (SU) that is responsible for: (1) the emulation of a global address space; (2) the communications through the network; (3) the inter-fiber synchronization; and (4) the implementation of a load balancing mechanism. When the model is implemented

[1] The only limitations on the number of active threads in a fine grain multi-threading system are caused by the memory space available to store active thread descriptors. If the data structure that holds these descriptors is stored in virtual memory, a very large number of active threads can indeed be supported.

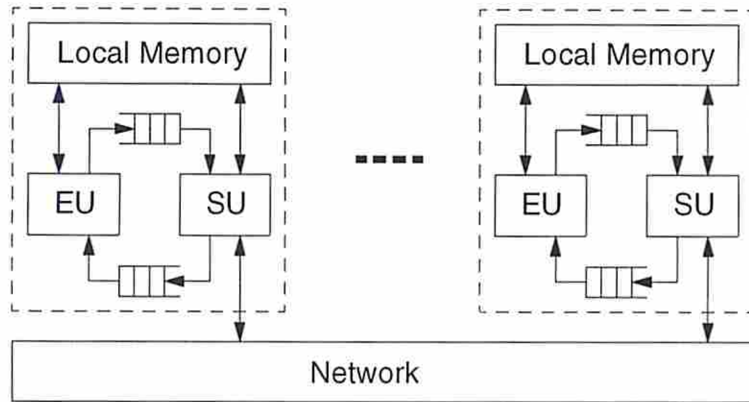


Figure 4.1: The EARTH Model

on processors with a single processor per processing unit, the functions of the SU are emulated in software by a RunTime System (RTS).

4.1.2 Split Phase Communication and Synchronization

A cornerstone of the EARTH model is the mechanism that enables the superposition of local computation and remote operations: the *split-phase transaction.* Whenever an operation involves a long and/or unpredictable latency, the statement that requests that the operation be performed is issued in one fiber and the statement that depends on the result of the operation is issued on a different fiber. A dormant fiber receives synchronization signals from other fibers — executing either in the same processor or on a remote processor — through a synchronization slot.

A typical split phase operation, an EARTH block-move-sync operation, is illustrated in Figure 4.2. In Figure 4.2(a): (1) a fiber running on the execution unit of processor P_i issues a request that a block of data be copied from the memory of a processor P_j to its local memory. The requesting fiber may continue performing operations that do not depend on the arrival of the requested block, but will eventually terminate and allow the EU of processor P_i to run other enabled threads. The block move request must specify the source and destination addresses for the movement

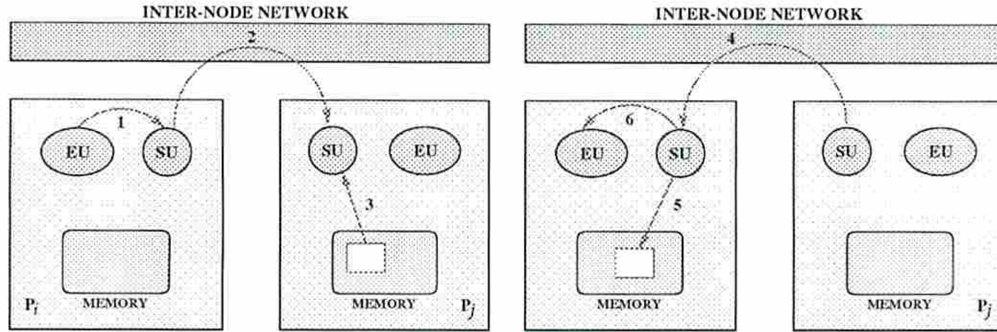


Figure 4.2: (a) (1) An active fiber in the EU of P_i requests an EARTH split-phase block-move-sync operation; (2) The SU of P_i decodes the source address to the memory of P_j and sends a request for the block; (3) The SU of P_j receives the request and reads the block from the local memory. (b) (4) The SU of P_j sends the block over the network to the SU of P_i; (5) The SU of P_i writes the block in the local memory; (6) The SU of P_i decrements a synchronization slot counter, that becomes zero and causes the spawning of a fiber that will use the block transferred.

as well as the address of a synchronization slot that will receive a synchronization signal when the data transfer is complete. (2) The SU of P_i, having received the request for the block move, sends a block request to the SU of P_j through the network. (3) the SU of P_j reads the requested block from the local memory of P_j. In Figure 4.2(b): (4) the SU of P_j sends, through the network, the requested block to the SU of P_i. (5) The SU of P_i writes the block into the destination address. Finally (6) the synchronization slot indicated in the block move request receives a synchronization signal and causes the fiber that will use the transferred data to be spawned and executed in the EU of processor P_i. In this example we assume that the destination of the block move and the synchronization slot that received the signal upon the completion of the data transfer were in the same processor that requested the data movement. However the EARTH model is general enough to allow each one of these addresses to be in a different processor.

Observe in the example presented in Figure 4.2 that the EU of processor P_j is never involved in the data transfer requested by processor P_i. Thus if two processing units are actually available in the machine to support the EU-SU model of EARTH,

the only impact of the data movement on the execution of fibers in P_j would be possible conflicts on accesses to the memory between the SU and the EU. Moreover, during the steps (2) to (6) in Figure 4.2, the EU of P_i is not involved and is free to execute other enabled thread. The capacity to overlap the remote data transfer with the execution of other fibers in the EU is a distinguishing characteristic of a fine grain multi-threading system.

4.2 Single Assignment Storage Structures

In this chapter we study the use of software cache for I-structures, a single-assignment data structure, in the EARTH model. The name *I-structure* was originally used by Arvind and Thomas in the context of functional languages to designate an array built with a fine-grained update operator with no repeated indexes [11]. Later, I-structures were proposed as separate data structures for functional programs. In [10], Arvind, Nikhil, and Pingali demonstrate, through several programming examples, that the introduction of I-structures in functional languages eliminates inefficiencies and increases the programmability of functional languages. The proposition to incorporate I-structures in functional languages was derived from the observation that without the ability to store a *state*, it is very difficult to solve even simple problems in a manner that is efficient, easy to code, and enables the exploitation of parallelism [10].

Our motivation to introduce a single assignment structure in Threaded-C stems from the observation that the use of such structures significantly reduces the number of synchronization operations required in some programs. The single-assignment characteristic of I-structures eliminates the need for consistency related network operations when these structures are enhanced with temporary storage buffers. The former makes it easier to code problem solutions in Threaded-C, and the latter makes it easier to implement software caches for I-structures.

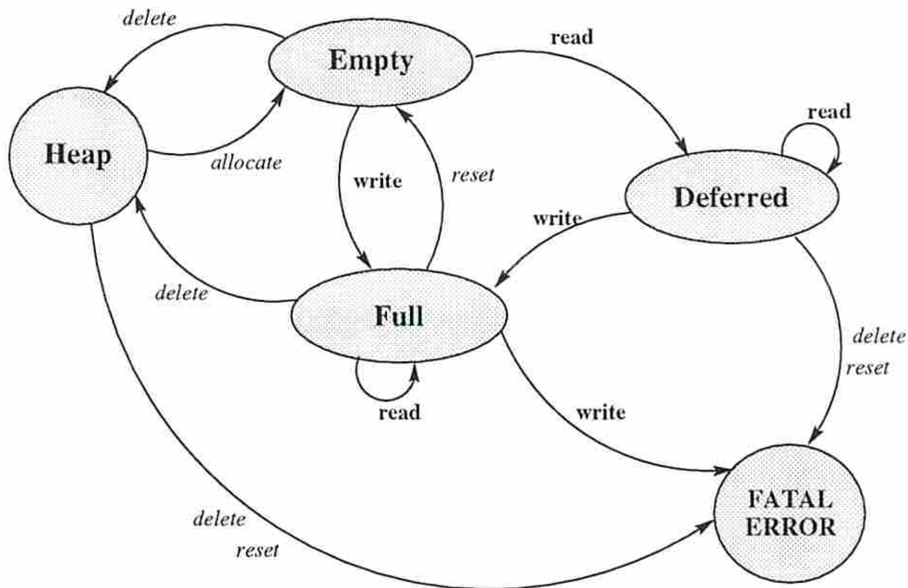


Figure 4.3: State Transition Diagram for the I-Structure Implementation

Originally an I-structure was defined as an array of elements [2], where each element of the array can be in one of three states: *empty*, *full*, and *deferred*. Each element of the array can only be written once, thus the name *single-assignment*, but it can be read many times. When the I-structure is created, all the elements of the array are empty. If a read occurs before the write, the element goes into the deferred state and the read operation is kept in a queue associated with that element. Subsequent reads are also queued. When a write to an empty element occurs, the value is written and the element becomes full. If the element was in the deferred state, all the reads that were queued for that element are serviced before the writing operation is complete, and the element goes into the full state. A read to a full element returns immediately with the value previously written. A write to a full element is considered a *fatal error* and causes the program to terminate. Figure 4.3 shows the state diagram of an I-structure. Notice that this state diagram includes the operations

[2]However, nothing prevents the implementation of a single element I-structure, or other data structure organizations.

delete and *reset* that were not in the original definition of an I-structure. These operations were included in our implementation because, different from the functional language environment in which the I-structure was originally defined, Threaded-C is an imperative language that does not offer garbage collection. Therefore the programmer must delete data structures after they are no longer needed. The reset operation allows reuse of I-structures avoiding frequent deletion/allocation in some applications

Observe that for its proper functioning, the state transitions on the I-structure must be atomic. For instance if a write is performed in a deferred element, all reads in the queue of the deferred element must be served the value written before another operation to the same element can be performed. In the current implementations of Threaded-C this atomicity is derived from the fact that fibers are non-preemptive and that with a single processor in each processing node, only a single thread can run on a node at a time.

The two key functions to implement I-structures in threaded-C are the I-READ and I-WRITE operations.

THREADED I_READ_x(int iid, int index, void *GLOBAL place, SPTR slot_adr)

Reads the element `index` of the I-structure `iid`. The value read is written in `place` by a split phase transaction that when completed synchronizes the slot `slot_adr`. If the element `index` is empty, `I_READ` stores `place` and `slot_adr` in the reading queue corresponding to that element. When the write operation to that element is performed, the value written is copied in `place` and the slot `slot_adr` is synchronized.

THREADED I_WRITE_x(int iid, int index, T value)

Writes `value` to the element `index` of the I-structure `iid`. If the element `index`

is full, `I_WRITE` prints a fatal error message in the standard error output and terminates the program.

4.3 ISSC Implementation on EARTH

4.3.1 ISSC implementation using Threaded-C language

Split-phased transactions for remote data memory accesses provide the ability to tolerate communication latency in a multi-threaded system. The data obtained through a split phase transaction is managed by the programmer, and is not automatically cached by the system. Therefore if repeated requests for the same data are issued, they will be sent through the network to the source of the data requested.

We presented our design and implementation of our I-Structure Software Cache (ISSC) in Chapter 3 [51, 52, 53] to cache I-Structure elements on multi-threaded systems that support split-phased transactions. The ISSC takes advantage of the spatial and temporal localities of memory operations in I-Structure memory systems.

The single assignment property of the I-Structure memory system enables the implementation of the ISSC as a *software cache* without any hardware support. The ISSC intercepts all the read operations to the I-Structure. A remote memory request is sent out to the remote host only if the requested data is not available on the software cache of the local host. We explore the spatial locality in the references to the I-structure through a blocking mechanism. Instead of requesting a single element of the structure, an entire block of data including the requested element is requested to the node that hosts the I-structure.

The state transition diagram for an element of the ISSC is shown in Figure 4.4. There is no space allocated in the ISSC for *invalid* elements. An invalid element might be allocated in ISSC and change state because a read of the element is performed by the local node, or because a read to another element in the same block is performed. In either case a request is issued to the host node. If the element

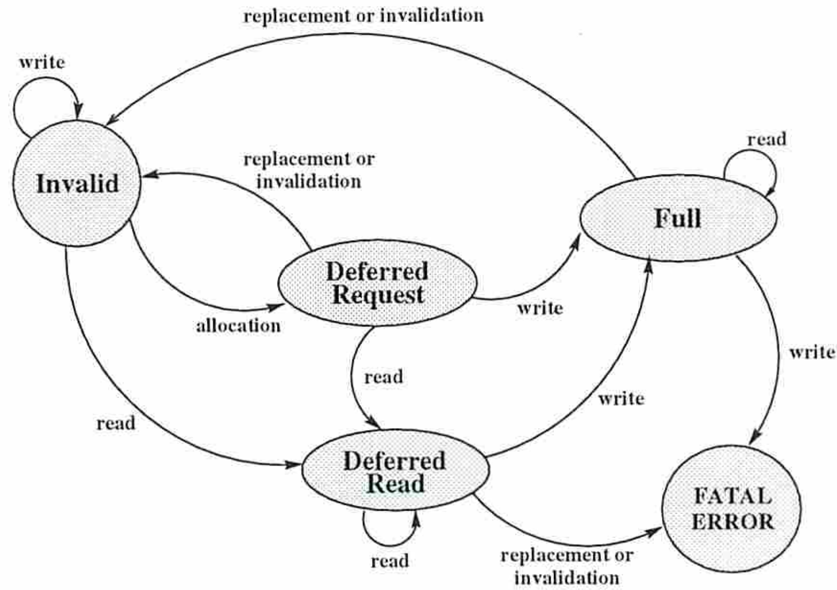


Figure 4.4: State Transition Diagram for the I-Structure Software Cache

itself is read, it goes into the *deferred read* state, otherwise it goes into the *deferred request* state. If a read to a deferred request element is issued, there is no need for a new request to be issued and the element goes into the deferred read state. Further read operations to a deferred read element are queued in the element and do not cause further state transitions. If a write to a deferred read or deferred request is performed in the host node and the value written is sent to the local node, the element goes into the *full* state. Read operations for elements in the *full* state are serviced immediately and do not cause any state transition. A write to a full element is a fatal error. Both a full element and a deferred request element can be evicted from the ISSC either by a replacement operation or by an invalidation operation. A deferred read element is *irreplaceable*. An invalidation or a replacement of such an element is a fatal error. A write to an invalid element is ignored and the element is not placed in the cache.

The *ISSC* is implemented in the Threaded-C [6, 70] language for EARTH [37] systems. Our implementation of ISSC builds on the I-Structure user library [7, 5]. In

this section we describe the key data structures, functions and policies implemented in the ISSC library.

DATA STRUCTURE *Cache*.

This is the main data structure for I-Structure software caches. The layout of our software cache is the one of a set-associative cache. Set-associative software caches have faster cache entry searching time than fully associative caches and better cache utilization than direct mapped caches. The caching address consists of the node number of the host node, the I-Structure I.D. and the index of the element for which a read is requested. Upon receiving a read request, the caching address is mapped to a set by a hash function, and a software search is performed to see if there is a match for the address in the set. In our simulation studies [51, 52], we determined that a cache block of 8 data elements would yield reasonable cache hit ratio. Therefore, in our experiments discussed in chapter 5, we use a cache block size of 8 and implemented the software cache with 256 sets and 8 cache blocks within each set. That would be $16K$ elements in the cache. The complete definition of the data structures used in ISSC implementation is shown in Appendix A.

THREADED InitCache(SPTR done)

InitCache allocates memory space for software cache in local node and initializes it. The initialization should be done before any cache accesses. After the initialization, a synchronization signal is sent to the address `done`.

THREADED SC_I_READ(int node, int i_id, int index, int type, void *GLOBAL place, SPTR slot_adr)

This is the read function for I-Structure elements through the utilization of the software caches. Instead of invoking the original I_READ_X at the remote node in which the I-Structure is allocated to request an I-Structure data element, the SC_I_READ is invoked in the local node. No requests are sent to the owner

node of the I-Structure if the data already exists in the local software cache or if the element has already been requested. The node that hosts the I-Structure is `node_id`, `i_id` is the I-Structure requested, `index` is the element of the I-Structure, `type` is the data type of the element, and `place` and `slot_adr` are the address where the requested data will be sent and synchronized when the data is back.

4.3.2 Usage of ISSC in Threaded-C language

A simple example to show how the ISSC library is used in a Threaded-C program is shown in Figure 4.5. In this example, an I-Structure floating point array of length 8 is allocated on the last node of the system. The data of these I-Structure elements are then generated by a node, and node 0 reads back the value of those 8 data elements.

In line 4, we define the I-Structure host, I_NODE, as the last node of the system, NUM_NODES-1. In Thread_0 (lines 20-25), we initialize the I-Structure in I_NODE and software caches on each node. In Thread_1 (lines 27-31), a floating point I-Structure array of 8 elements is allocated in I_NODE. The handle for the allocated I-Structure is stored in F_str. In Thread_2 (lines 33-45), the data of I-Structure array *F_str* are generated by *ARRAY_INIT* function (line 6-12) which is invoked by *TOKEN* function in line 34. Then, data of this I-Structure array are read back in line 35-44. In line 38, we use a compiler flag to activate/deactivate the ISSC. If the *CACHE* flag has been defined in the compilation, the function *SC_I_READ* is invoked locally (at node NODE_ID), otherwise, the function *I_READ_F* is invoked in the I-Structure host (node I_NODE). After all the 8 data elements are read back from node I_NODE, THREAD_3 is activated, prints out the data, and terminates the program. In this program, if ISSC is not used, 8 data requests for the I-Structure array F_str are sent to the remote node, I_NODE, by invoking 8 I_READ_F functions

```
1:  #include <stdio.h>
2:  #define EXTERN
3:  #include "issc.h"
4:  #define  I_NODE   NUM_NODES-1
5:
6:  THREADED ARRAY_INIT(int i_node, int i_id, int length)
7:  {
8:     int i;
9:     for(i=0; i < length; i++)
10:       INVOKE(i_node, I_WRITE_F, i_id, (float)i);
11:    END_FUNCTION();
12: }
13:
14: THREADED MAIN()
15: {
16:    SLOT SYNC_SLOTS[3];
17:    int F_str, i;
18:    float F_variable[8];
19:
20:    INIT_SYNC(0,NUM_NODES+1,NUM_NODES+1,1);
21:    INVOKE(I_NODE, I_INIT, SLOT_ADR(0));
22:    /* Allocate cache space on each node */
23:    for(i=0; i<NUM_NODES; i++)
24:       INVOKE(i, InitCache, SLOT_ADR(0));
25:    END_THREAD();
26:
27:  THREAD_1:
28:    INIT_SYNC(1, 1, 1, 2);
29:    /* Allocate I-Structure */
30:    INVOKE(I_NODE, I_ALLOCATE, 8, TO_GLOBAL(&F_str), SLOT_ADR(1));
31:    END_THREAD();
32:
33:  THREAD_2:
34:    TOKEN(ARRAY_INIT, I_NODE, F_str, 8);
35:    INIT_SYNC(2, 8, 8, 3);
36:    /* Read from F_str[0:7]*/
37:    for ( i=0; i<8; i++)
38: #ifdef CACHE
39:          INVOKE(NODE_ID, SC_I_READ, I_NODE, F_str, i, F,
40:                    TO_GLOBAL(&F_variable[i]), SLOT_ADR(2));
41: #else
42:          INVOKE(I_NODE, I_READ_F, F_str, i,
43:                    TO_GLOBAL(&F_variable[i]), SLOT_ADR(2));
44: #endif
45:    END_THREAD();
46:
47:  THREAD_3:
48:    for(i=0; i<8; i++) printf("%f ", F_variable[i]);
49:    RETURN();
50: }
```

Figure 4.5: Threaded-C with ISSC program example

on I_NODE. However, if ISSC is used, even though 8 SC_I_READ functions are invoked locally, only one data request is sent to the remote node I_NODE.

A more complete example of using ISSC in a real application written by Threaded-C language is shown in Appendix B.

Chapter 5

Experiment results on EARTH systems

To study the effectiveness of our implementations of both I-structures and ISSC, we coded four benchmarks (see Section 5.3) in both versions of the system (Threaded-C with I-Structure and Threaded-C with I-Structures and ISSC). Our experimental results were obtained in the MANNA machine. We also measured the latency for basic EARTH operations and for I-structure based operations.

5.1 Highlights of Experimental Results

Our main results can be summarized as follows:

1. **The addition of ISSC to the EARTH system results in increased robustness to latency variation.** The speedup obtained with ISSC increases for machines with higher costs for remote operations (see Figures 5.1 and 5.2 for details).

2. **The ISSC significantly reduces the amount of traffic in the network.** As shown in Table 5.3 in all applications the number or remote requests for I-structure elements was reduced from one up to four orders of magnitude.

3. **The sole addition of I-Structures (without ISSC) decreases the performance of the EARTH system.** Even for machines with higher latencies, the overhead of the software emulation of I-structures hurts performance

(as shown in the graphs of Figure 5.3) unless it is offset by the benefits of the ISSC (see Section 5.3 for details).

4. **ISSC operations can be implemented very efficiently in the MANNA machine.** In 5.2 we demonstrate that the MANNA machine network interface is very efficient. Our experiments demonstrated that our implementation of the ISSC on top of the EARTH operations is also efficient (see Table 5.1).

5. **The performance of the system with ISSC improves for all benchmarks for machines with moderately high latency for remote operations.** As shown in Figure 5.2 for all four benchmarks if 10 μs (500 cycles) are added to the latency of MANNA (which is 3.5 μs = 175 cycles), the benchmarks running on the software with the ISSC produces greater spreedup over the system with I-structures only.

5.2 The Cost of ISSC Operations

Our studies are based on an implementation of EARTH on the MANNA machine. MANNA is a 20 node, 40 processor machine. Each node has two Intel i860 XP processor running at 50 MHz with 32 MB memory and is interconnected with other nodes through a crossbar switch network. The MANNA machine is a research platform of which only a few were constructed. With the full control of network interface in MANNA machine, the implementations of inter-node communication and synchronizations are very efficient as demonstrated by the measurements presented in this section. We measure the latency of some EARTH and ISSC operations for the EARTH-MANNA-SPN machine. EARTH-MANNA-SPN is an implementation of the EARTH model on the MANNA machine in which only one processor is used in each node [69].

Operation	Local	Remote
Get_Sync	141	348
Fun. Call	250	451
I_READ_F	317	492
ISSC hit	479	—
ISSC miss	2693	—
ISSC deferred	1354	—

Table 5.1: Latency of EARTH and ISSC operations on EARTH-MANNA-SPN, measured in number of cycles (1 cycle = 20 ns).

The MANNA machine is a research platform of which only a few were constructed. With the full control of network interface in MANNA machine, the implementations of inter-node communication and synchronizations are very efficient as demonstrated by the measurements presented in this section. However, this network efficiency is usually not available in affordable and widely available networks of workstations. The sending and receiving of network packages may take from hundreds to thousands of cycles depending on the design of the network interface [21]. In some machines, a parallel environment is built on top of the TCP protocol and the communication interface overhead may be as high as hundreds of micro-seconds [44]. Even with some improved protocols, like Fast Sockets [66] and Active Messages [72], it still costs 40~60 micro-seconds to send a message to the network.

The latency of the operations required to communicate and synchronize across processing nodes is a determinant factor in the performance of some applications. Observe that the processor is not busy with the operation for the number of clock cycles shown in Table 5.1. Most of the remote operation time is spent either waiting on queues or in the network, thus releasing the processor to execute other ready fibers.

Table 5.1 lists the latency of some EARTH and ISSC operations in the MANNA platform used in the analytical model. In a local measurement all operations are within a processor, while in a remote measurement, all operations are issued to other

nodes through network. The EARTH operations measured in Table 5.1 include a `get_sync` operation in which thread 1 requests a word of data from thread 2 and thread 2 synchronizes thread 1 when the data arrives; and function calls which represent the invocation of a threaded function either in the same node or in a remote node.

At the bottom of Table 5.1 are the measured latency of ISSC operations and of the basic I-Structure read function, `I_READ_F`. The measurement starts from thread 1 invoking the I_READ_F function in I-Structure node either in the same node or in a remote node until the I_READ_F function finished and synchronizing thread 1 when the data arrives. `ISSC hit` measures the invoking of an I_READ_F for a remote data, finding the requesting data in local software cache and synchronizing the requesting thread with the data found in software cache. `ISSC miss` is the case that the entire surrounding data block is not found in the software cache and a new request for the whole block is issued to a remote node, and finally the requested data along with the whole data block are sent back from remote node and the synchronization is done. Notes that, this measurement is made by issuing multiple requests in a pipeline fashion. Therefore the time spent on the remote node is overlapped with other issues of requests and only the time spent in local node is measured. `ISSC deferred` is the case that the surrounding data block already allocated in the local software cache however the requested data element is not there yet. The original request is therefore deferred in the software cache until the requested data is available along with entire data block or sent back individually from remote I-Structure node. The same measurement as *ISSC miss* is done to ensure that no idle time and remote operation time is measured.

The difference between local and remote cases of I_READ_F denotes four times of the communication interface overhead. It includes one for the requester sending the request, one for the I-Structure node receiving the request, one for I-Structure node sending the data back and finally one for the requester receiving the data. The

one-way communication interface overhead takes only 175/4 processor cycles (0.825 μs). This measurement indicates that the inter-node communication in MANNA machines is very efficient when compared with network of workstations.

5.3 Description of Benchmarks

To measure the improvement in the system performance when both I-structures and ISSC are used, we selected four different benchmarks: dense matrix multiplication, Conjugate Gradient, Hopfield network, and sparse matrix multiplication. To compare the performance of the software cache with the original system, we implemented three versions of codes for each benchmark: A *plain Threaded-C* code, a Threaded-C code using the I-Structure library, *Threaded-C+IS* and a Threaded-C code using both the I-structure library and the I-Structure Software Cache (*ISSC*), *Threaded-C+ISSC*. All our experimental results were performed in the MANNA machine. The two processors of a processing node on MANNA share 32 Mbytes of DRAM. The nodes of MANNA are diskless, therefore all the code, runtime system, data, and the software emulations of the I-structure and the ISSC must fit in 16 Mbytes per node. Therefore we were only able to test moderate data set size for the benchmarks. In a related research work, Theobald developed a detailed cycle-by-cycle simulation of the MANNA architecture and demonstrated that applications scale well for larger versions of the platform [69].

Dense Matrix Multiplication. Two 128x128 dense matrices are multiplied. The algorithm that we use in this study is a simple minded, non-blocking algorithm that computes $C = A \times B$. The computation of rows in the resulting matrix C is evenly distributed among all nodes. Node 0 invokes threads on each node to compute the rows that it is responsible for. The results of C elements are written directly to where they reside.

Conjugate Gradient. The Conjugate Gradient algorithm from the NAS benchmark suite [13, 12] uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. In our experiment, the problem size is 256 linear equations with 256 unknown variables. Calculations of matrix-vector multiplications are done in parallel across all the nodes and the calculations of vector-vector multiplication are done on node 0. In this algorithm, most of the computation consists in updating array elements. Therefore, the benchmark does not have much temporal data locality.

Hopfield Network. Hopfield is a kernel benchmark [7] based on the Hopfield Network. It is a recursive neural network that is often used in combinatorial optimization problems as well as an associative memory. The network is formed by a set of neurons that are connected by synapses. At time $k + 1$, the activation value of each neuron is updated based on the activation values of neurons at time k weighted by synapse values. In the I-Structure and ISSC implementation, two I-Structure arrays are used to store the current and previous activation values of neurons. Before updating to the current value, the I-Structure array is reset and reassigned with a new I.D., therefore, the same memory space can be re-utilized and no cache flush would be needed. The problem size we tested is 256 neurons.

Sparse Matrix Multiplication. Sparse matrix multiplication is an application with irregular data access pattern. Two unstructured sparse 256x256 matrices A and B are randomly generated with density of 10%. Matrix A is then stored in Compress Row Storage (CRS) format and matrix B is stored in Compress Column Storage (CCS) format. A dense resulting matrix C is generated by multiplying A and B.

Number of Nodes	Benchmarks			
	Dense M.M.	C.G.	Hopfield	Sparse M.M
2	99.71	93.70	99.90	99.92
4	99.52	93.69	99.80	99.87
8	99.13	93.52	99.61	99.76
16	98.35	92.92	99.22	99.53

Table 5.2: I-Structure Software Cache Hit Ratios (%)

Number of Nodes	Benchmarks							
	Dense M.M.		C.G.		Hopfield		Sparse M.M	
	no ISSC	w/ISSC	no ISSC	w/ISSC	no ISSC	w/ISSC	no ISSC	w/ISSC
2	528384	1536	33536	2112	32768	32	986668	761
4	396288	1920	25152	1587	24576	48	731842	971
8	231168	2016	14672	950	14336	56	426002	1038
16	123840	2040	7860	557	7680	60	227979	1078

Table 5.3: Average number of remote memory requests per node

Table 5.2 shows the cache hit ratios of the four benchmarks in our experiments and Table 5.3 shows the average number of remote memory requests in each benchmark both without and with ISSC. ISSC did help the system to exploit global data locality. For three of the four benchmarks (except conjugate gradient), more than 99% of cache hit ratios could be achieved, and even in conjugate gradient algorithm which has poor temporal data locality, 93% of cache hit ratio could be achieved. Table 5.3 shows that ISSC reduces the number of remote memory requests actually sent to remote nodes. In all the cases, at least 93% of the original remote memory requests are eliminated out by the I-Structure Software Cache.

5.4 Robustness to Latency Variation

We measured the speedup between the I-Structure Software Cache version of the benchmarks and a version of the same benchmarks written in plain Threaded-C and running on a single processing node. We performed two sets of experiments. The first set, shown in Figure 5.1 measures the performance on the MANNA machine. As a

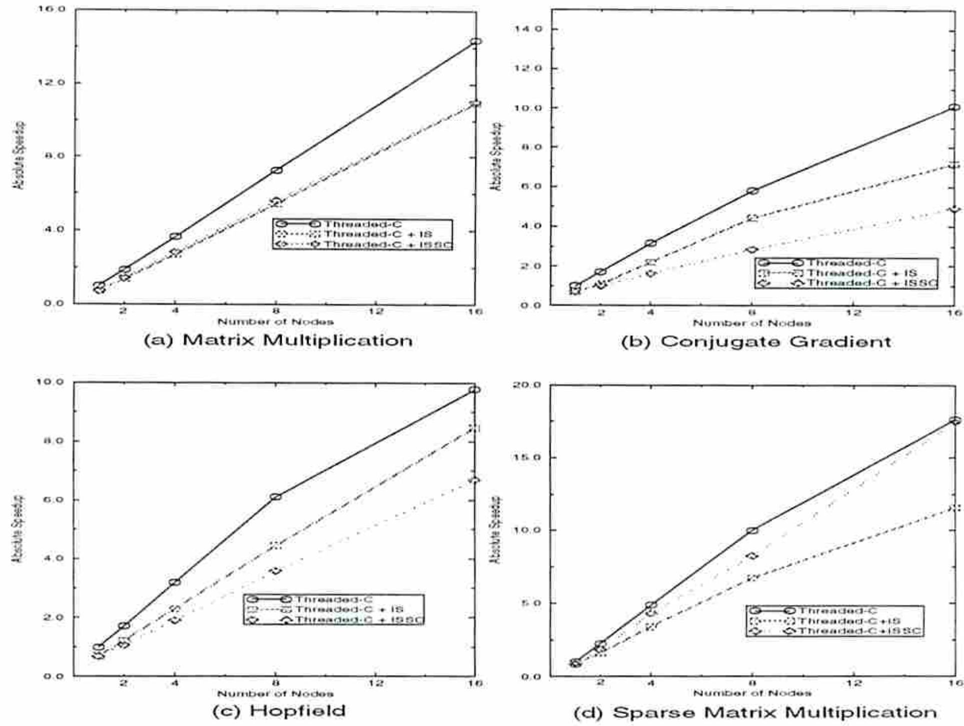
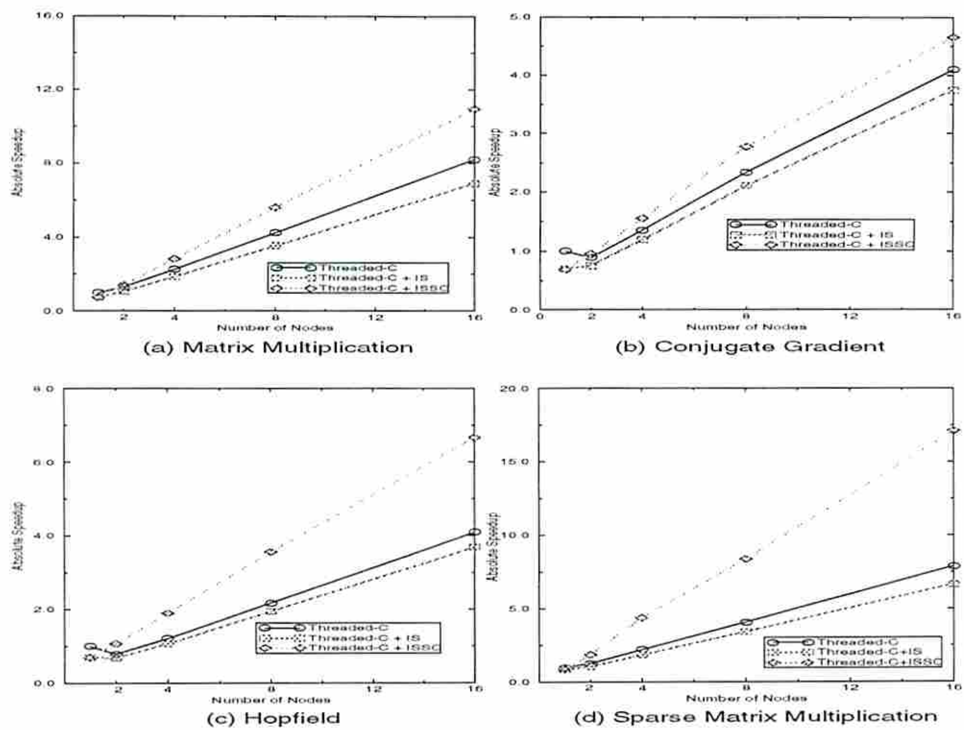


Figure 5.1: Speedup in the MANNA machine.

result of the efficient implementation of the network and its interface on the MANNA machine, the *plain Threaded-C* version have the best performance for all benchmarks. When the cost to execute split-phase operations is very low, the overhead incurred in the I-Structure and software cache operations in *Threaded-C+IS* and *Threaded-C+ISSC* can degrade the performance. In the Conjugate Gradient and Hopfield benchmarks, *Threaded-C+IS* version has better performance than the *Threaded-C+ISSC* version. This is because of the poor temporal data locality in the algorithms which results in the high ratio of deferred cache hits in the cache hit situation. The overhead of deferred hits is much larger than the I-Structure access and the ISSC hit as reported in the beginning of this section.

In our second set of experiments, shown in Figure 5.2, we add 10 μs to the cost of both I-structure and ISSC operations. This is equivalent to a machine with a



(a) Matrix Multiplication (b) Conjugate Gradient

(c) Hopfield (d) Sparse Matrix Multiplication

Figure 5.2: Absolute speedup with 10 μs communication interface overhead

higher communication/computation cost ratio, i.e., a machine in which requesting a remote

of four applications on the MANNA machine with 10 μs add-on synthetic communication interface overhead. In this set of experiments, *Threaded-C+ISSC* version out-performs other two versions in all applications. Even though we added 10μs synthetic communication interface overhead into this set of experiments, it is still far less than the communication interface overhead of fast local area network [66], which cost 40$\tilde{6}$0 micro-seconds. However, the *Plain Threaded-C* version still has better performance than *Threaded-C+IS* because of the I-Structure access overhead. In these experiment, we show that even though ISSC pays extra overhead for its operations, by taking advantage of the global data locality in applications and with the amount of communication interface overhead saved by ISSC, the I-Structure Software Cache improves system performance in the Network of Workstation platforms.

From the previous experiments, we know that the communication interface overhead is a determinant factor in the performance of I-Structure Software Caches. To have a better understanding of the relationship between the ISSC performance and the communication interface overhead, we ran our experiments on a 16 node system with a variable synthetic communication overhead for our selected benchmarks. Figure 5.3 shows the execution time of applications under different overhead. In each application, we marked the point of communication interface overhead where the *Threaded-C+ISSC* version starts to out-perform the *plain Threaded-C* implementation. ISSC starts to help the system when the communication interface overhead is greater than 6.3μs, 9.2μs, 6μs and 3.2μs respectively in dense matrix multiplication, conjugate gradient, Hopfield and sparse matrix multiplication. When the communication interface overhead exceeds 100μs, the *Threaded-C+ISSC* versions run almost 10 times faster than the *plain Threaded-C* versions in our benchmarks.

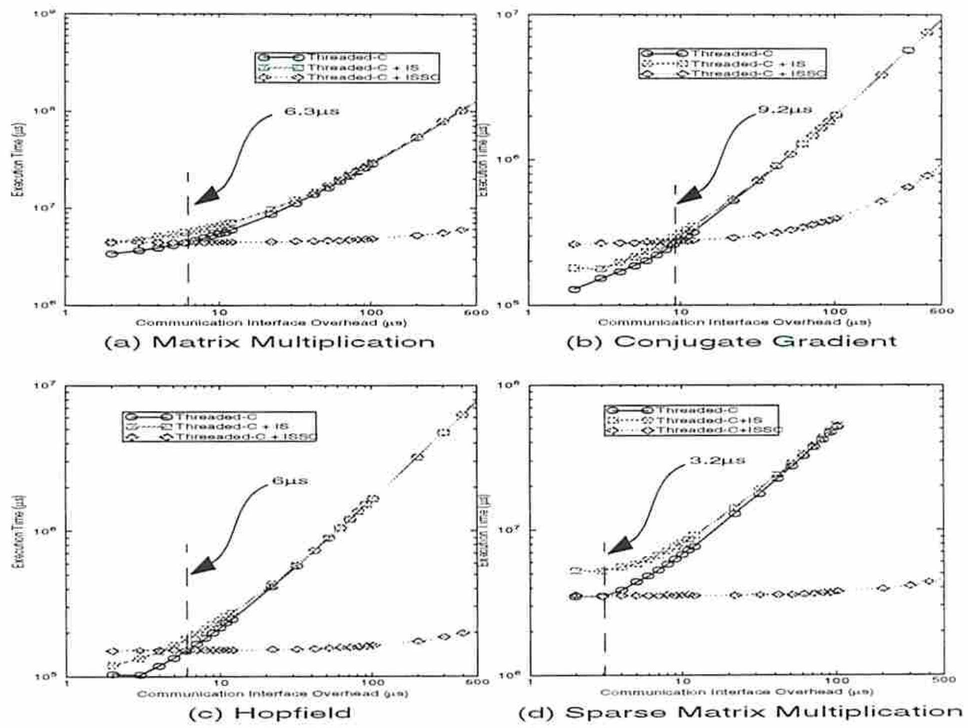


Figure 5.3: Execution time with synthetically variable communication interface overhead

5.5 Summary

In this Chapter, we compare the performance of two extensions to the original programming environment in the EARTH system: (i) the programming environment is extended with an implementation of I-structures, a single assignment data structure that facilitates the implementation of synchronizations across multiple processing nodes; (ii) the programming environment is extended with I-structures and an I-Structure Software Cache (ISSC) that enables the exploitation of temporal and spatial locality in the I-structures. The motivation to introduce both I-structures and ISSC to the original EARTH programming environment stems from the single assignment nature of I-structures. When single-assignment storage structures are used, the need for consistency related transactions in the network is eliminated.

Our studies are based on an implementation of EARTH on the MANNA machine. MANNA is a 20 node, 40 processor machine. Each processing node has two processors. The nodes are interconnected through a crossbar switch network. In the EARTH-MANNA version that we use, the functions of the synchronization unit are emulated in the same processor that executes the fibers. Because neither MANNA nor the original programming environment for EARTH provide direct support for single assignment storage such as I-structures, we emulate both the I-structure operations and the ISSC in software. Our study focuses on the robustness of the resulting programming model to latency tolerance. Therefore we measure the latency for basic EARTH operations, I-structure operations, and ISSC operations. Then we vary these latencies by introducing delays in the operations to identify the lower bound of latency (measured in processor cycles) beyond which the introduction of I-structure and ISSC in the system would no longer have a positive effect on performance. Our results indicate that the extension of the Threaded-C programming environment with ISSC is robust to variations on latency. This robustness is reflected in better speedup curves for machines with highes costs for remote operations.

Chapter 6

Performance Modeling

Our ISSC is a pure software approach to exploit the global data locality in non-blocking multithreaded execution without adding any hardware complexity. It provides the ability to reduce the communication latency while maintaining the ability to tolerate the communication latency in multithreaded execution. Some reasonable research questions are: *Do software caches really work? Will the overheads of software cache operations compromise its performance? What are the conditions for ISSC to improve system performance? What kind of applications could benefit from ISSC ?* It is the single assignment property of the I-Structure memory system that makes the use of a *software cache* profitable. Because the cache of a single assignment memory is inherently coherent, no cache coherence problem is encountered in an I-Structure cache design. The absence of coherence related operations significantly reduces the overhead of the software cache system. Indeed, with the capability of communication latency tolerance in multithreaded execution, the major benefit of ISSC comes from the saving from communication interface overhead.

In this Chapter, we present an analytical model for the performance of a multithreading system with and without ISSC support. This model consists of two sets of factors, platform-related and benchmark-related parameters, which affect the

performance of ISSC. From this model, we could analyze the lower bound of communication interface overhead from which ISSC starts to yield performance gain in different benchmarks and platforms.

6.1 Performance Analysis

Before we present the analytical model, we analyzes the performance measurement that we shown in Chapter 5. In table 5.2, we shown the cache hit ratios of the four benchmarks in our experiments and Table 5.3 shows the average number of remote memory requests in each benchmark both with and without ISSC. ISSC did help the system to exploit global data locality. For three of the four benchmarks (except conjugate gradient), more than 99% of cache hit ratios could be achieved, and even in conjugate gradient algorithm which has poor temporal data locality, 93% of cache hit ratio could be achieved. Table 5.3 shows that ISSC reduces the number of remote memory requests actually sent to remote nodes. In all the cases, at least 93% of the original remote memory requests are eliminated out by the I-Structure Software Cache.

With the capability of communication latency tolerance in multithreaded execution, the major benefit of ISSC comes from the saving from the communication interface overhead. In the measurements presented in Section 5.2 we show that the network implementation of MANNA is very efficient. However, this efficiency is usually not available in affordable and widely available networks of workstations. To have a better understanding of the relationship between the ISSC performance and the cost of remote operations, we ran our experiments on a 16 node EARTH-MANNA system with adding a variable synthetic communication interface overhead on top of existing overhead.

Figure 6.1 shows the execution time of applications as the communication overhead increases. The marked points are the values measured from actual runs on

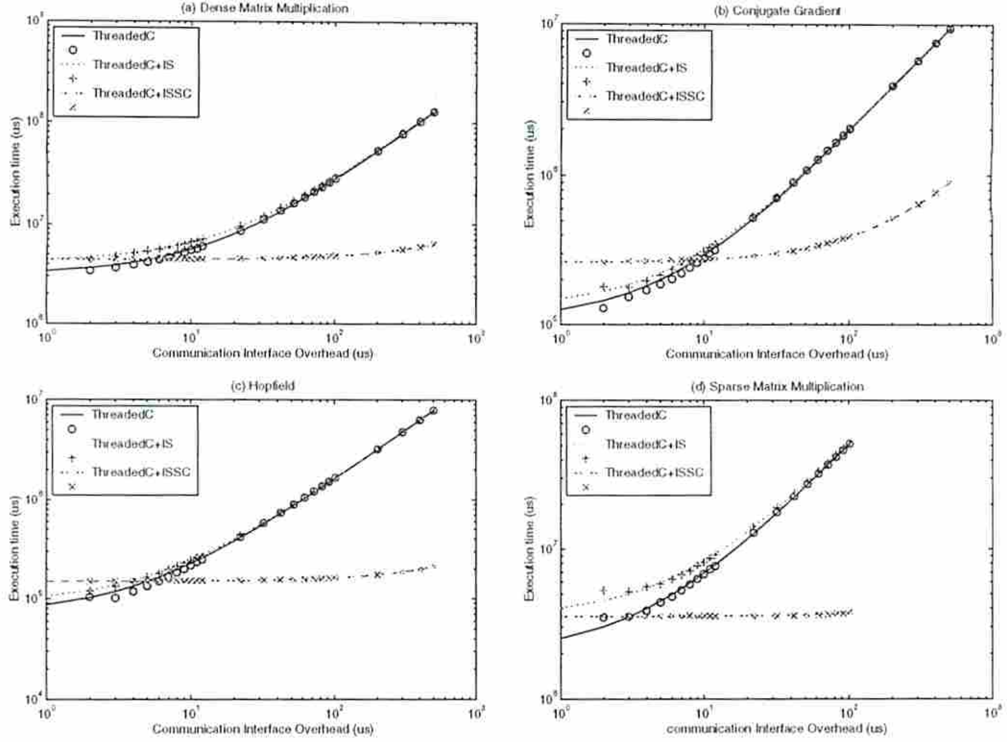


Figure 6.1: Execution time with add-on synthetically variable communication interface overhead. (a)Dense Matrix Multiplication (b)Conjugate Gradient (c)Hopfield (d)Sparse Matrix Multiplication

EARTH-MANNA machine and the timing curves are derived from the measurement in a least square sense with degree of 1. The timing equations are shown in Table 6.1. To find where ISSC starts to improve system performance, we could just solve the equation $T_{Threaded-C+ISSC} < T_{Threaded-C}$ to find the cross points of the $T_{Threaded-C+ISSC}$ and $T_{Threaded-C}$ curves. These points are shown at the right of Table 6.1. The logical meaning of these points is that when the communication interface overhead of a system is greater than the value plus the existing communication interface overhead on the MANNA machine (0.825 μs), then ISSC will yield increased performance gain. As we can see, ISSC starts to help the system when the communication interface overhead is greater than 7.1μs, 9.7μs, 5.9μs, and 4.8μs, respectively in dense matrix multiplication, conjugate gradient, Hopfield and sparse

Benchmarks	Versions			Cross Point
	Threaded-C	Threaded-C+IS	Threaded-C+ISSC	
Dense M.M.	$2.478 \times 10^5 C_{oa} + 3.139 \times 10^6$	$2.471 \times 10^5 C_{oa} + 4.174 \times 10^6$	$3877.6 C_{oa} + 4.431 \times 10^6$	5.3
C.G.	$1.855 \times 10^4 C_{oa} + 1.074 \times 10^5$	$1.847 \times 10^4 C_{oa} + 1.301 \times 10^5$	$1274.8 C_{oa} + 2.618 \times 10^5$	8.9
Hopfield	$1.548 \times 10^4 C_{oa} + 7.198 \times 10^4$	$1.541 \times 10^4 C_{oa} + 9.114 \times 10^4$	$118.8 C_{oa} + 1.505 \times 10^5$	5.1
Sparse M.M.	$4.855 \times 10^5 C_{oa} + 2.040 \times 10^6$	$4.811 \times 10^5 C_{oa} + 3.542 \times 10^6$	$1798.7 C_{oa} + 3.521 \times 10^6$	3.95

Table 6.1: Timing equations and the cross-points (μs)

matrix multiplication, which are still far less than the ones in most of network of workstations.

6.2 The Analytical Models

The experiments presented in Chapter 5 provide useful information about the performance of the ISSC on existing hardware platform. However we would like to be able to predict, for machines yet to be built, under which circumstances the implementation of ISSC becomes profitable. To enable such predictions, in this section, we develop an analytical model for the execution time of benchmarks written in Threaded-C, $T_{threaded-c}$, Threaded-C with I-Structure support, T_{IS}, and Threaded-C with both I-Structure and I-Structure Software Cache support, T_{ISSC}. The base for our analytical model is T_B, the execution time of the benchmark on a fine grain multi-threaded machine without I-Structures and the cost of split-phase memory accesses is deducted. Our model uses the following set of *benchmark-related parameters* and *platform-related parameters*:

Benchmark-related parameters:

- N_L: Number of local reads
- N_R: Number of remote reads
- R_{hit}: Cache hit ratio on remote reads
- R_{d-hit}: Cache deferred hit ratio

Platform-related parameters:

- C_o: One-way communication interface overhead (original)

C_{oa}:	One-way communication interface overhead (add-on)
O_I:	Local I-Structure read service time
O_r:	Read request invoking time
O_{hit}:	ISSC hit service time
O_{miss}:	ISSC miss service time
O_{def}:	ISSC deferred hit service time

Where R_{d-hit} is the ratio between the cache hits that have been deferred and the total number of cache hits. The higher R_{d-hit} is, the poor temporal data locality is in the application. The C_o and C_{oa} are defined as one-way communication interface overheads which are only incurred in either sending or receiving network data, but not both. The definitions of other platform-related parameters were presented in Section 5.2. In our analytical model O_{miss} does not include communication interface overhead.

The analytical models are defined as follows:

$$T_{threaded-c} = T_B + (N_L + N_R)O_r + N_R 2(C_o + C_{oa})$$

$$T_{IS} = T_B + N_L O_I + N_R O_r + N_R 2(C_o + C_{oa})$$

$$T_{ISSC} = T_B + N_L O_I + N_R R_{hit}(1 - R_{d-hit})O_{hit} + N_R R_{hit} R_{d-hit} O_{def} + N_R(1 - R_{hit})O_{miss} + N_R(1 - R_{hit})2(C_o + C_{oa})$$

In the development of the analytical model, we assume owner computation rule. Therefore, all the write operations are performed locally and incur no communication overhead. We also assume that the I-structure arrays are evenly distributed across the nodes. Therefore that the jobs are also evenly distributed. We assume the same basic execution time, T_B for all three versions of the system. In fact, T_B in T_{ISSC} should be smaller than the ones in $T_{threaded-c}$ and T_{IS} because caching remote memory requests decreases the average turn-around time for all the requests and as a result, it increases parallelism and processor utilization. However, this assumption

Parameters	Benchmarks			
	Dense M.M.	C.G.	Hopfield	Sparse M.M
N_L	139328	614	512	19140
N_R	123840	9210	7680	234784
R_{hit} (%)	98.35	93.65	99.22	99.55
R_{d-hit} (%)	20.00	51.80	100.00	21.20

Table 6.2: Benchmark-related Parameters

Parameters	C_o	O_I	O_r	O_{hit}	O_{miss}	O_{def}
micro-second	0.875	6.34	2.82	9.58	51.54	27.08

Table 6.3: Platform-related Parameters Measured from MANNA machine

in T_{ISSC} provides the upper-bound of the execution time for the system with ISSC. In our implementation, only remote reads are cached in ISSC. Hence, those local I-Structure reads in T_{ISSC} still need the I-Structure read service in local node. In these models, the remote costs for $T_{threaded-c}$ and T_{IS} are $N_R 2(C_o + C_{oa})$ and for T_{ISSC} is $N_R(1 - R_{hit})2(C_o + C_{oa})$ which only include the communication overhead incurred in the local node. The overheads in remote node are actually hidden by the multithreaded execution.

6.2.1 Verifying the Model

To verify the analytical models, we compare the execution time prediction obtained from the models with our experimental results on EARTH-MANNA shown in Chapter 5. In Table 6.2, we list the benchmark-related parameters which are collected from our experiments on MANNA for a selected set of benchmarks. The platform-related parameters of MANNA machine, measured in Section 5.2, are listed in terms of μs in Table 6.3.

From our analytical models, we know that the execution time of *Threaded-C* and *Threaded-C+IS* versions are linear proportional to the add-on communication overhead, C_{oa}, with the factor of 2 times number of remote reads, $2N_R$, which are 247680, 18420, 15360, and 469568 respectively in dense matrix multiplication,

conjugate gradient, Hopfield and sparse matrix multiplication. Also, the execution time of *Threaded-C+ISSC* is also linear proportional to C_{oa} with the factor of two times the number of cache misses, $2N_R(1 - R_{hit})$, which are 4086, 1170, 119, and 2113 respectively. These numbers match the curve-fitting timing equations from our experiments described in Table 6.1 within 10% error range.

According to the analytical models, for $T_{ISSC} < T_{threaded-c}$, we need,

$$(N_L + N_R)O_r + N_R 2(C_o + C_{oa}) > N_L O_I + N_R R_{hit}(1 - R_{d-hit})O_{hit} + N_R R_{hit} R_{d-hit} O_{def} + N_R(1 - R_{hit})O_{miss} + N_R(1 - Rhit)2(C_o + C_{oa})$$

$$\Rightarrow \frac{N_R}{N_L+N_R} R_{hit}(2C_o + 2C_{oa}) > \frac{N_L}{N_L+N_R} O_I + \frac{N_R}{N_L+N_R} R_{hit}((1 - R_{d-hit})O_{hit} + R_{d-hit} O_{def}) + \frac{N_R}{N_L+N_R}(1 - R_{hit})O_{miss} - O_r \ldots\ldots\ldots..(1)$$

The meaning of Equation 1 is quite straight forward. The condition for ISSC starts to improve the system is that the communication interface overhead saved by ISSC (left hand side of the equation) should be greater than the I-Structure read service time required for local access plus ISSC operation overhead minus the read request handling time in the original system (right hand side of the equation). We plug in the N_L, N_R, R_{hit}, and R_{d-hit} parameters for each benchmark and use the MANNA parameters to derive the minimum add-on communication interface overhead from which point ISSC starts to improve the system performance. We get 6.7μs, 9.0μs, 11.5μs, and 4.6μs respectively for dense matrix multiplication, conjugate gradient, Hopfield, and sparse matrix multiplication.

Our analytical model for T_{ISSC} defines the upper bound of the execution time. Therefore, the cross-point derived from Equation 1 is the lower-bound of communication interface overhead from which ISSC starts to improve system performance. For example, if the point derived from our models is 10μs, for this upper-bound estimation of T_{ISSC}, we could say that as long as the communication interface overhead is larger than 10μs, our ISSC is going to improve the performance. Values of these cross-points derived from our analytical models are greater than but close to

the values we measured in our experiments shown in Table 6.1 except in Hopfield. This is because the synchronization of the activation updates after each time stamp yields partial sequential behavior. In this case, the basic execution time in T_{ISSC} is much smaller than in $T_{threaded-c}$. Therefore the cross-point we predicted is much larger than what we measured.

6.2.2 Performance Predictions

In this section, we introduced our analytical models for the multithreading system with and without I-Structure software cache support and we verified these models with our experiment results based on EARTH-MANNA machine. With these models, we could predict the lower bound of communication interface overhead from which ISSC starts to yield performance gain in different kind of benchmarks and platforms.

By using these models, for a fixed platform parameters (like plug in the parameters measured from EARTH-MANNA) and varied benchmark-related parameters, we could estimate the value of communication overhead where Threaded-C+ISSC starts to out-perform pure Threaded-C for the benchmarks with different characteristics. Figure 6.2 shows these cross-points of different kinds of benchmarks by varying the cache hit ratio and deferred hit ratio while assuming only half of memory requests are issued to remote nodes. From this figure, we could see that even in those benchmarks with poor locality ($R_{hit} = 0.5$ and $R_{d-hit} = 1.0$), ISSC still yield performance gain for communication interface overhead greater than $40\mu s$, which is still faster than most of the network implementation in network of workstations. For those benchmarks with extremely good locality, *i.e.* more than 98% of cache hit ratio with 0 deferred hit, ISSC starts to improve the system for the communication overhead as low as $5\mu s$.

Some researcher dedicate their work on communication optimizations to reduce the number of remote memory accesses. This kind of optimizations are based on

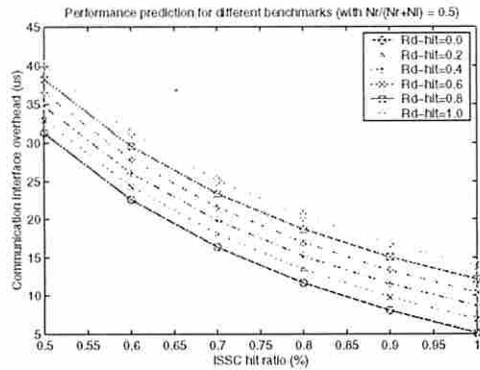


Figure 6.2: Performance prediction for different benchmarks

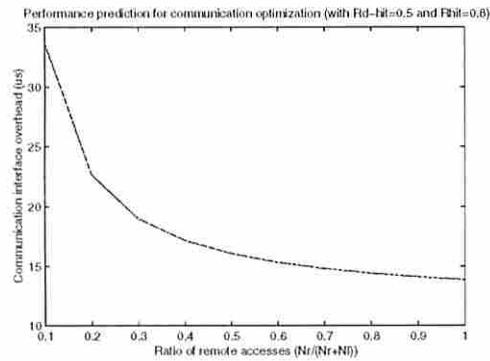


Figure 6.3: Performance prediction for communication optimization

the static analysis of the program behavior which is different from exploiting the data locality during the run-time by the caches. However, ISSC could still yield performance gain in the benchmarks compiled with the communication optimization techniques. In Figure 6.3, we vary the ratio of remote memory requests to the total number of memory requests. We find out that even in an application with only 10% of memory accesses are remote and moderate cache hit ratio ($R_{hit} = 0.8$ and $R_{d-hit} = 0.5$) ISSC still improves the system at $33.5\mu s$ of communication overhead.

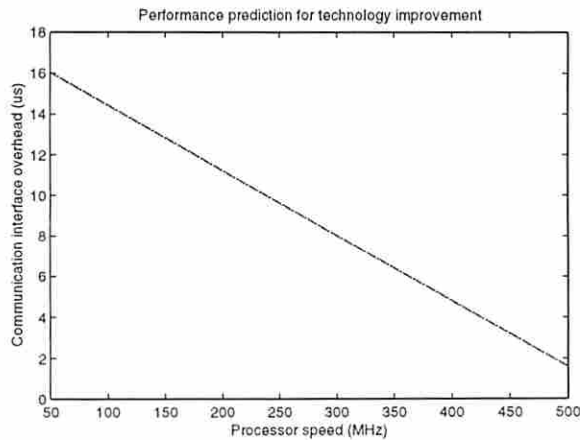


Figure 6.4: Performance prediction for technology improvement

As the speed of processors becomes faster and faster, the gap between the computation and communication latencies become larger and larger. Because, our ISSC is a pure software implementation, the ISSC operation overhead decreases proportional to the increase of processor speed. In Figure 6.4 we vary the platform-related parameters based on 50MHz MANNA processor by increasing the speed of processors for an application with 50 % of remote memory accesses, 80% cache hit ratio, and 50% deferred hit ratio. From this curve, we could predict that if we have a 500MHz processor available, which is already there on the market, the cross-point drops to less than $2\mu s$. In this case, ISSC could almost yield performance gain on any parallel machine.

6.3 Summary

Do software caches really work? In this chapter, we demonstrated a software implementation of I-Structure cache, *i.e. ISSC*, can deliver performance gains for most distributed memory systems which don't have extremely fast inter-node communications, such as network of workstations [21, 44, 66, 41].

ISSC caches values obtained through split-phase transactions in the operation of an I-Structure. It also exploit spatial data locality by clustering individual element requests into block. Our experiment results show that the inclusion of ISSC in a parallel system that provides split-phase transactions reduces the number of remote memory requests dramatically and reduces the traffic in the network. The most significant effect to the system performance is the elimination of the large amount of communication interface overhead which is incurred by remote requests.

We developed analytical models for the performance of a distributed memory multithreading system with and without I-Structure Software Cache support. We verified these models with our experiment results on an existing multithreaded architecture, EARTH-MANNA. These models consist of two sets of factors, platform-related and benchmark-related. Platform-related parameters are those latencies incurred by remote memory requests and ISSC operations. Benchmark-related parameters are the characteristics of applications, such as number of remote and local memory accesses and data locality. By finding the cross-point of two execution time curves, which have the communication interface overhead as variable, of the systems without and with ISSC, we could find when ISSC starts to yield performance improvement for different benchmarks and platforms. Through systematic analysis, we show that ISSC delivers performance gains for a wide range of applications in most of the parallel environments, especially in network of workstations.

Chapter 7

Conclusions and future research

7.1 Conclusions

In this dissertation, a split-phased transaction caching scheme for the I-Structure-like memory systems is proposed and implemented as a runtime system to exploit global data locality in the non-blocking multithreaded systems. Our ISSC provides a software caching mechanism to further reduce the communication latency by caching the split-phase transactions while maintaining the benefits of latency tolerance in multithreaded execution.

The ISSC design was first validated by our Generic MultiThreaded machine (GMT) simulator with several benchmarks. Then, we implemented our ISSC as an user library on EARTH systems using Threaded-C language. With the implementation on real machines, we were able to measure the overhead of the ISSC operations and measure its actual performance with some benchmarks. We further developed analytical models for the for the performance of a multithreading system with and without ISSC support. From these models, we can analyze the lower bound of communication interface overhead from which ISSC starts to yield performance gain in different benchmarks and platforms.

The following contributions are achieved in this research work,

- Combination of the benefits of latency tolerance and latency reduction in distributed memory multiprocess systems. Traditional multithreading models provide the capability of latency tolerance through overlapping useful computation with the long communication overhead in distributed memory environment. *Caching* provides the capability of this latency reductions in the shared memory environment. However, our ISSC provides a software caching mechanism to further reduce the communication latency by caching the split-phase transactions while maintaining the benefits of latency tolerance in multithreaded execution in distributed memory multiprocessor systems.

- Network traffic reduction to reduce communication overhead and network contention. From our experiments, we shown the effect of our ISSC on network traffic reduction. More than 90% of the original remote memory request are eliminated out by our ISSC. Each remote memory request needs to be sent through network interface to the remote node, and each request will suffer from the network interface overhead four times. Therefore, our ISSC eliminates quite large amount of network interface overhead incurred by remote memory requests. Moreover, it relieves network traffic and avoids potential network contention problems.

- Harmless low-cost software implementation. ISSC is a pure software approach to exploit the global data locality with adding any hardware complexity. The design of ISSC is efficient enough to be implemented in software layer without degrading the system performance. Indeed, the overhead of ISSC itself would had dragged down the system performance, but the tremendous amount of communication interface overhead saved by the ISSC not only compensate its overhead but also improve the overall system performance.

- Single thread performance improved by latency reduction. In some applications with embarrassing parallelism, the long communication latency may not

be tolerated by enough threads. In these applications, ISSC's capability of latency reduction could improve the system performance.

- Consistent cache performance and robust fine-grain multi-threaded execution in Network of Workstation platform. The cache advance scheme in our ISSC provides the adaptability to the unpredictable communication characteristics in the Network of Workstation environments. This makes the system achieve the same performance without being affected by the variation of the communication latency. ISSC also eliminate tremendous amount network interface overhead incurred by the large number of split-phase remote memory requests in the fine-grain multithreaded systems. This make the fine-grain multithreaded execution more robust in the NOW platforms.

- Frame work for further split-phased transaction cache design. This research established a solid foundation for further split-phased transaction cache design. The design issues we discussed and the approaches adopted by our design provide fundamental knowledge for it. The analytical model we developed would allow us to predict the performance of the caching with advanced technology improvement which may not be available today.

7.2 Future research

There are several research directions could be derived from this research.

- Cache coherence protocol design and implementation for multiple-assignment split-phased transactions. The ISSC could be extended with cache coherence protocol when multiple-assignment storage systems are required. In some application, frequent updates of variables are desired and using I-Structures in this kind of application may degrade the system performance because of excessive overhead caused by frequent I-Structures deallocation and reallocation.

Extending the split-phased transaction software caches with proper coherence protocols could do a lot of help on exploiting global data locality for all application. Releasing the constraint of memory construct and destruct of I-Structures would provide the programmers with full control of memory usages and hence programmers have more flexibility to implement application.

However, the extra overhead incurred by the cache coherence protocol needs to be evaluated in more details. It may make the software caches less beneficial because of heavier software cache operation overhead. Fortunately, releasing the single assignment constrain in memory construct will simplify the cache design in some aspects. For example, no more deferred reads on data elements and therefore no deferred read handling is needed. Further more, the whole memory block could be brought back to the requester without checking the states of each individual data elements. All of these may still make the idea of split-phased transaction software cache for multiple assignment memory system feasible and beneficial.

- Hardware supports for I-Structure caching. Hardware supported cache systems for split-phased transactions could further manifest the benefits of I-Structure caching in non-blocking multithreaded architectures. There are two different approaches to implement the hardware supported cache: The first approach is to use a piece of *dedicated hardware*. The concept of I-Structure caches is not limited on software implementation. It could be implemented in hardware as well. With a customized chip or FPGA to work as the controller of I-Structure cache management along with SRAM for caching data storage, the overhead of I-Structure cache operations could be reduced dramatically, and hence, I-Structure cache will deliver performance gains and more significant performance improvement on all platforms. Indeed, the I-Structure memory system management could be also incorporated into this dedicated hardware

to further improve the system performance. The alternative of hardware supported I-Structure caching is to use a *decoupled processor*. A decoupled processor for communication and memory management has been adopted in many multiprocessor system design. The operations of I-Structure memory access and caching could be also executed on this decoupled processor. This will offload the I-Structure cache overhead from other processors which then could be dedicated to useful computation. As a matter of fact, no complex floating point operations needed in these management jobs, and therefore, a low cost micro-controller or DSP could be used as this purpose.

- Network Caching. While some researchers concentrate on the development of faster network interfaces [16, 27, 57], the concept of our split-phased transaction caches for the distributed data could be integrated into next generation network interface design. In this network interface, a message from local processor requesting for a remote data element will be translated into a new message requesting for the whole data block containing the original requested element. A cache block space will be reserved for the new request in the network interface before the message is injected into the network and the successive requests for the elements in this block will wait in the interface without actually been sent to the remote nodes. The requested remote data element along with other surrounding elements in the same data block are cached in the interface when they are brought back from remote nodes. Cache coherence protocol should be also implemented in the design to provide general purpose usages for parallel computing on NOW platforms. Using this next generation network interface with the capability of network caching in NOW, network traffic could be reduced dramatically and fine-grain parallelism in NOW platforms would become possible.

- Integrating Data Caches into Non-Strict Caches. The hardware supported I-Structure cache could be further integrated with local L2 caches. With this approach, the remote data fetched via split-phased are only stored in the integrated cache. There are no local storages for remote data. Therefore, all the data references are referred by the global addresses as in shared memory systems.

  In this approach, the continuation vector carried by split-phased fetch only includes the thread number which is going to consume this data without providing local storage address for storing the fetched data. When data is brought back to local host and stored in the integrated non-strict data cache, a signal is sent to consuming thread to inform the arrival of requested data item. When all the data become available in the non-strict data cache, it is enabled and the data are accessed directly from caches during the execution.

  To avoid the cache controller automatically replace the data that was just fetched before the corresponding thread is actually executed, we could eliminate this problem by allowing memory reservations in the local cache. When a cache line is first allocated for a missing read, the cache line is reserved and the missing read is deferred at the corresponding location for the requested element. The deferred read is pending on the data cache until the data item is brought back from remote node and is referred by the consuming thread. To implement this, when a remote data item arrives and is stored into the cache, if the cache element is in *deferred* state, signals are sent to the consuming threads indicating by the pending reads to inform the arrival of this data item. However, these pending reads are not de-queued until the data item actually referred by the consuming threads. When the consuming starts to execute and refers the data item, the pending read associated with this consuming thread is removed from the queue. The cache element is not changed to *present* state until all the pending reads are removed. As long as there is any *deferred* cache

element in the cache line, the cache line will still be reserved until all the cache elements in this cache line are all either in *present* or *empty* state. A reserved cache line is not subject to replacement by the cache controller until its state has been changed to non-reserved by the runtime system. Our initial studies indicate that this property can be integrated with the support for non-strict access hardware cache with the single addition of one bit for the state representation of each cache line.

- Apply non-blocking multithreaded execution model with split-phased transaction cache support on SMT architectures. A robust fine-grain non-blocking multithreaded execution model with ISSC support could be implemented on versatile architectures. It would be very interesting to implement it on SMT architectures [71, 54, 32, 15]. Each ready thread from the execution model has all the variables it needs locally and is guaranteed to be run from start to the end without synchronization, remote memory requests and other long latency operations inside the thread. Each ready thread is an independent entity and won't interfere with each other. This would very likely to drive the SMT processors with very high throughput. Since all the variables needed in a thread are available locally, we could further bring all the data frame memory and instruction frame memory into caches right before the thread is scheduled for execution. All of these features of non-blocking multithreaded execution model applied on SMT architectures could fully exploit the benefits of SMT architecture for a single application performance.

Reference List

[1] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, September 1992.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *ISCA 95*, 1995.

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B.H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[4] Rovert Alverson, David Callahan, Daniel Cummings, Brain Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International conference on Supercomputing*, June 1990.

[5] J. N. Amaral, G. Gao, and X. Tang. An implementation of a hopfield network kernel on earth. In *X Brazilian Symposium on Computer Architecture and High Perfor mance Processing*, pages 223–232, Buzios, RJ, Brazil, Sept. 1998.

[6] J.N. Amaral, Z. Ruiz, S. Ryan, A. Marques, C. Morrone, and G.R. Gao. Portable Threaded-C Release 1.1. Technical note 05, Computer Architecture and Parallel System Laboratory, University of Delaware, September 10 1998.

[7] Jose Nelson Amaral and Guang R. Gao. Implementation of I-Structures as a Library of Functions in Portable Threaded-C. Technical note 04, Computer

Architecture and Parallel System Laboratory, University of Delaware, July 28 1998.

[8] B. S. Ang, Arvind, and D. Chiou. StartT the Next Generation: Integrating Global Caches and Dataflow Architecture. CSG MEMO 354, Laboratory for Computer Science, MIT., February 1994.

[9] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, October 1989.

[10] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[11] Arvind and R. E. Thomas. I-structures: An efficient data structure for functional languages. Technical Report MIT/LCS/TM-178, Massachusetts Institute of Technology, Cambridge, 1981. MIT Lab. for Computer Science.

[12] D. Bailey, E Barszcz, K. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, March 1994.

[13] D. H. Bailey, J. T. barton, T. A. Lasinski, and H. D. Simon. The NAS parallel benchmarks. Technical Report NASA Technical Memorandum 103863, NASA Ames Research Center, July 1993.

[14] Michael J. Beckerle. Overview of the START(*T) multithreaded computer. In *Digest of Papers, 38th IEEE Computer Society International Conference, COMPCON Spring '93*, Feb. 1993.

[15] M. Bekerman and *et al.* Performance and hardware complexity tradeoffs in designing multithreaded architectures. In *Proceedings of Parallel Architectures and Compilation Techniques*, 1996.

[16] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, 1994.

[17] D. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.

[18] D. Cann and J. Feo. Sisal 1.2 : An Althernative to FOTRAN for shared Memory Multiprocessors. Technical Report UCRL-102263, Lawrence Livermore National Laboratory, 1989. rev 1. for ACM SIGGPLAN '90.

[19] D. Cann and R. Oldehoeft. A guide to the optimizing Sisal compiler. Technical Report UCRL-MA-108369, Lawrence Livermore National Laboratory, Sep. 1991.

[20] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 8–11, 1991.

[21] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[22] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. A compiler-controlled threaded abstract machine. In *Proceedings of ASPLOS-IV*, April 1991.

[23] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and T. von Eicken. Empirical study of a dataflow language on the CM-5. In G.R. Gao, L. Bic, and J-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 187–210. IEEE Press, 1994.

[24] F. Darema, D.A. Georage, V.A. Norton, and G.F. Pfister. A single program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, April 1988.

[25] J. B. Dennis and G. R. Gao. On Memory Models and Cache Management for Shared-Memory Multiprocessors. CSG MEMO 363, Laboratory for Computer Science, MIT., March 1995.

[26] Jack B. Dennis. The Paradigm Compiler: Mapping a functional language for the Connection Machine. In *Scientific Applications fo the Connection Machine*, pages 301–315, 1989.

[27] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Proceedings of the Hot Interconnects Symposium V*, August 1997.

[28] Michel Dubois and Faye A. Briggs. Effects of Cache Coherence in Multiprocessors. In *Proceedings of the* 9th *Annual Symposium on Computer Architecture*, pages 299–308, May 1982.

[29] Guang R. Gao. An Efficient Hybrid Dataflow Architecture Model. *Journal of Parallelism*, 19(4), December 1993.

[30] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong. Parallel Function Invocation in a Dynamic Argument-Fetching Dataflow Architecture. In *Proc. of PARBASE-90: Intl. Conf. on Databases, Parallel Architectures, and their Applications, Miami Beach, Florida*, pages 112–116, March 1990.

[31] J-L Gaudiot and C-T Cheng. A Scalable Cache Design for I-Structures. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1996.

[32] M. Gulati and N.Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of Int'l Symp. on High-Performance Computer Architecture*, 1996.

[33] Robert H. Halstead Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, 1988.

[34] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performamce Studies of Id on Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, pages 273–300, 1993.

[35] High Performance Fortran Forum. High-performance fortran language specification. Technical report, Rice University, May 1993.

[36] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for fortran d on mimd distributed-memory machine. In *Proceedings of Supercomputing '91*, pages 86–100, Nov. 1991.

[37] H. H.J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A Design Study of the EARTH Multiprocessor. In *PACT 95*, June 1995.

[38] Herbert Hing-Jing Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture*. PhD thesis, School of Computer Science, McGill University, Montreal, Québec, 1992.

[39] Robert A. Iannucci. *A dataflow/von Neumann Hybrid Architecture*. PhD thesis, Massachusetts Institute of Technology, July 1988.

[40] Robert A. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, May 1988.

[41] V. Karamcheti and A. Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of the 6th ACM International Conference on Architectural Support for Programming Languages and Systems (ASPLOS VI)*, Oct. 5-7, 1994.

[42] M. Katevenis. *Reduced instruction set computer architectures for VLSI*. PhD thesis, Comput. Sci. Division (EECS), UCB/CSD 83/141, Univ. California at Berkeley, Oct. 1983.

[43] K. M. Kavi, A.R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam. Design of Cache Memories For Multi-Threaded Dataflow Architecture. In *ISCA 95*, pages 253–264, 1995.

[44] K. Keeton, T. Anderson, and D. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III: A Symposium on High Performance Interconnects*, August 1995.

[45] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[46] Yuetsu Kodama and *et al.* A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation. In *Proceedings of InfoJapan 90*, pages 291–298, October 1990.

[47] Yuetsu Kodama and *et al.* EMC-Y: Parallel processing element optimizing communication and computation. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 167–174, July 1993.

[48] Charles Koelbel. Compuile-time generation of communications for scientific programs. Technical report crpc-tr91089, Center for Research on Parallel Computation,Rice University, January 1991.

[49] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing '88*, pages 28–34, Nonvember 1988.

[50] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *ISCA 94*, 1994.

[51] Wen-Yen Lin and Jean-Luc Gaudiot. I-structure Software caches - A split-phase transaction runtime cache system. In *Proceedings of the 1996 Parallel Architectures and Compilation Techniques Conference*, Oct. 1996.

[52] Wen-Yen Lin and Jean-Luc Gaudiot. Exploiting Global Data Locality in Non-Blocking Multithreaded Architectures. In *Proceedings of the Third International symposium on Parallel Architectures, Algorithms and Networks*, Dec. 1997.

[53] Wen-Yen Lin and Jean-Luc Gaudiot. The Design of An I-Structure Software Cache System. In *Workshop on Multithreaded Execution, Architecture and Compilation, 1998. Held in conjunction with HPCA-4*, Feb. 1998.

[54] M. Loikkanen and N. Bagherzadeh. A fine-grain multithreading superscalar architecture. In *Proceedings of Parallel Architectures and Compilation Techniques*, 1996.

[55] O. C. Maquelin, H. H.J. Hum, and G. R. Gao. Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors. In *Proceedings of EURO-PAR'95*, August 1995.

[56] J. R. McGraw and *et al.* SISAL: Streams and iteration in a single assignment language - language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[57] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, 1996.

[58] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computer? In *Proceedings of ISCA-16*, May-Jun 1989.

[59] Rishiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. CSG MEMO 305, Computation Structures Group, 1990.

[60] H. Nishikawa, H. Terada, S. Komori, K. Shima, T. Okamoto, and S. Miyata. Architecture of a VLSI-Oriented Data-Driven Processor: the Q-v1. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice Hall, 1991.

[61] Michael D. Noakes, Deborah A. Wallah, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, May 1993.

[62] G.M. Papadopoulos. *Implementation of a General-purpose Dataflow Multiprocessor*. PhD thesis, Laboratory for Computer Science, MIT., August 1988.

[63] G.M. Papadopoulos. *Implementation of a General-purpose Dataflow Multiprocessor*. The MIT Press, 1991.

[64] G.M. Papadopoulos and D. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the* 17th *Annual International Symposium on Computer Architecture*, pages 82–91, June 1990.

[65] D. Patterson and C. Sequin. A VLSI RISC. *IEEE Computer Mag.*, 15(9):8–21, Sept. 1982.

[66] S. Rodrigues, T. Anderson, and D. Culler. High-Performance Local Area Communication With Fast Sockets. In *USENIX 1997 Annual Technical Conference*, Jan 1997.

[67] L. Roh and W. A. Najjar. Design of Storage Hierarchy in Multithreaded Architectures. In *IEEE Proceedings of MICRO-28*, pages 271–278, 1995.

[68] Shuichi Sakai and *et al.* An architecture of a dataflow single chip processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, May 1989.

[69] Kevin B. Theobald. *EARTH - An Efficient Architecture for Running THreads.* PhD thesis, School of Computer Science, McGill University, Montreal, Québec, 1999.

[70] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the portable threaded-c language. CAPSL Technical Memo 19, University of Delaware, http://www.capsl.udel.edu, March 16 1998.

[71] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simulatneous multithreading: Maximizing on-chip parallelism. In *22nd Ann. Int'l Symp. On Computer Architecture*, 1995.

[72] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 19–21, 1992.

[73] X3J3. *FORTRAN 90, draft of the inetnational standard.* The FORTRAN Technical Committee of ANSI, 1990.

[74] N. Yoo. Generic MultiThreaded machine (GMT) simulator. Computer engineering technical report, Department of Electrical Engineering - Systems, University of Southern California, December 1993.

[75] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, January 1988.

Appendix A

ISSC's Implementation on EARTH using Threaded-C Language

The complete definitions of data structures for I-Structure Software Caches (ISSC) implementation is defined in *"issc.h"* header file. ISSC operations are also defined in that header file.

A.1 ISSC Structure

```
/* include the data structures defined for i-struct */
#include "i-struct.h"

/* total of 16K words by default */
#define CacheBlockSize 8
#define CacheSetSize 8
#define NofCacheSet 256

int NI_DELAY;  /* parameter for add-on network interface delay */
int NET_LATENCY; /* parameter for add-on network latency */
int CNofRead;
int CNofHit;
int CNofDeferred;
int CNofRemoteHit;
int CNofRemoteMiss;

/********** Data Structure definition for Software Cache ************
*  Cache Block, is the basic unit of Cache access.
*              "g_array", is the array ID (now using the beginning
*                         address of the referred I-Structure array.
*              "b_index", is the index of first cache element in
```

```
 *                          the original I-Structure array.
 *                "handler", the the array which stores the sync. slot
 *                          of deferred request service handler for
 *                          each element to handle the pending
 *                          requests.
 *                In a cache block,
 *                   If ( (deferred_flag==0) && (tag==NUL) ) {
 *                        means an empty cache block
 *                   } else if ((deferred_flag==1) && (tag!=NULL)) {
 *                        means the request of this cache block is on
 *                        going.
 *                   } else if ((deferred_flag==0) && (tag!=NULL)) {
 *                        means this cache block is a valid cache
 *                        block.
 *                   } else means in error state
 *
 *                   If (reserved==1) {
 *                        means that this cache block has at least one
 *                        deferred request, so that it can not be
 *                        replaced.
 *                   else means this cache block is free to be
 *                        replaced.
 *
 *  Cache Set, contains the index of next victim when cache replace
 *              is needed.
 *
 *  Cache, contains several performance measurement variables and
 *          an array of CacheSet.
```

```
*         The location of CacheSet to be checked is indexed by a
*         simple hash function.
*****************************************************************/
typedef struct CacheBlock_str CacheBlock;
struct CacheBlock_str {
  char deferred_flag;
  char referenced;
  char type;
  unsigned long g_array;
  int  b_index;
  array_cell element[CacheBlockSize];
  SPTR handler[CacheBlockSize];
};

typedef struct CacheSet_str CacheSet;
struct CacheSet_str {
  int victim;
  CacheBlock block[CacheSetSize];
};

typedef struct Cache_str Cache;
struct Cache_str {
  int NofRead;
  int NofLocal;
  int NofHit;
  int NofDeferred;
  int NofInitMiss;
  int NofReplaced;
```

```
  int NofPassed;
  int NofWrite;
  int NofRemoteHit;
  int NofRemoteMiss;
  double hit_time;
  double miss_time;
  double tag_time;
  double service_time;
  CacheSet set[NofCacheSet];
};

enum {
  B=0,
  S,
  L,
  F,
  D,
  G,
  BL
};

/* Software Cache space is allocated here */
Cache *cache;
```

A.2 ISSC Operations

The following functions define ISSC operations either invoked by user program or invoked with ISSC operations.

```
/* Flushing ISSC */
THREADED FlushCache(SPTR done);

/* ISSC initialization function */
THREADED InitCache(int ni_delay, SPTR done);


/* These threaded functions should be invoked locally */

/* I-Structure element fetch using ISSC */
THREADED SC_I_READ(int i_node, int iid, int index, int type,
     void *GLOBAL place, SPTR slot_adr);

/* I-Structure block fetch using ISSC */
THREADED SC_I_READ_BLOCK(int i_node, int iid, int index,
                         int block_size, void *GLOBAL place,
                         SPTR slot_adr);

/* This function is called by software cache library to be invoked
   in I-Structure node */
THREADED I_BLKMOV_RSYNC(int iid, int index, void *GLOBAL c_block,
void *GLOBAL place, int block_size,
unsigned long g_array, SPTR slot_adr);
```

```
THREADED I_BLKMOVBLOCK_RSYNC(int iid, int index,
                                void *GLOBAL c_block,
     void *GLOBAL place,
                                void*GLOBAL data_buf,
     int block_size,
     unsigned long g_array, SPTR slot_adr);

THREADED Block_Handle(int i_node, int iid, int index, int type,
      int set_no, int block_no, int element_no);

THREADED Block_Handle_BLOCK(int i_node, int iid, int index, int
    block_size, int set_no, int block_no,
    int element_no);

THREADED Single_Handle(int type, int set_no, int block_no,
                          int element_no, SPTR done);

THREADED Deferred_Server(int type, int set_no, int block_no, int
         element_no);

THREADED Deferred_Server_BLOCK(int block_size, int set_no,
                                 int block_no, int element_no);

THREADED Or_Deferred_Server(int type, int set_no, int block_no,
    int element_no);

THREADED Or_Deferred_Server_BLOCK(int block_size, int set_no,
```

```
                                        int block_no, int element_no);

/* The following functions are used to add the network interface
   overhead */
THREADED BLOCK_I_READ(int i_node, int iid, int index, int type,
      void *GLOBAL place);

THREADED NOW_I_READ(int i_node, int iid, int index, int type,
      void *GLOBAL place, SPTR slot_adr);

THREADED NOW_I_READ_TEST(int i_node, int iid, int index, int type,
 void *GLOBAL place, SPTR slot_adr);

THREADED NOW_I_WRITE_B(int i_node,int iid, int index,
       char value);

THREADED NOW_I_WRITE_S(int i_node,int iid, int index,
       short int value);

THREADED NOW_I_WRITE_L(int i_node,int iid, int index,
       long int value);

THREADED NOW_I_WRITE_F(int i_node,int iid, int index,
       float value);

THREADED NOW_I_WRITE_D(int i_node,int iid, int index,
       double value);
```

```
THREADED NOW_I_WRITE_G(int i_node,int iid, int index,
       void *GLOBAL value);

THREADED NOW_I_WRITE_BLOCK_SYNC(int i_node,int iid, int index,
void *GLOBAL origin, SPTR slot_adr);

THREADED NOW_GET_RSYNC(void *GLOBAL src, void *GLOBAL dest, int type,
       SPTR slot_adr);

void delay(int delay_par);

THREADED Print_Cache_Util(SPTR done);

THREADED Gather_Cache_Stat(int if_cache, int dim, int delay_time,
                           int exec_time);
```

Appendix B

Using ISSC with Hopfield Benchmark

The following program shows how Hopfield benchmark is implemented using Threaded-C with the support of I-Structure and ISSC. A makefile is also shown here to show how the program is compiled and how to use ISSC option.

B.1 Hopfield Benchmark

```
/*****************************************************************
*
* hopfield - An implementation of a Hopfield kernel in Threaded-C.
*
* Author: Jose Nelson Amaral <amaral@capsl.udel.edu>
*         Computer Architecture and Parallel Systems Laboratory
*         (http://www.capsl.udel.edu) - University of Delaware
*
* Purpose: Implement a solution for a Hopfield network kernel
*          demonstrating the use of the I-STRUCTURES in a
*          synchronization mechanism.
*
* Release Date: June 11 1998
*
*****************************************************************/

/*****************************************************************
*
* Revised by Wen-Yen Lin <wenyenl@usc.edu>
*
* Purpose: Implement a larger number of neurons suitable for testing
*          if I-Structure and I-Structure Software Caches.
```

```
*
* Release Date: April 29 1999
*
******************************************************************/

#include <stdio.h>

/*
 * Because the implementation of the I-structure library uses
 * conditional pre-processing, the user must include the empty
 * definition of EXTERN in the file that contains the MAIN function
 * before the file i-struct.h is included. This line must not be
 * present in any other files.
 */

#define EXTERN

#include "issc.h"
Cache *cache;



/* Cyclic 1 distribution macros */
#define OWNER(index) ((index) % NUM_NODES)
#define POSITION(index) ((index) / NUM_NODES)
#define IMAP(node, pos) ((pos)*NUM_NODES + (node))

/* Block distribution macros */
/*
```

```
#define OWNER(index) ((index) / ((DIM*DIM)/NUM_NODES))
#define POSITION(index) ((index) % ((DIM*DIM)/NUM_NODES))
#define IMAP(node, pos) ((node)*((DIM*DIM)/NUM_NODES) + (pos))
*/

#define STOPPING_CRITERIUM  0.0001

int NET_SIZE;

/*
float synapse[NET_SIZE];
*/
float *synapse;

/* I-Structure IDs for neuron activations */
float *i_old, *i_new, *temp;

float change_of_state;

/* Declare 2 arrays here for storing the activation */
float Array1[256];
float Array2[256];

/*********************************************************************
*
* Adds the value of all the changes of state in each neuron, and
* synchronizes the slot specified by {\tt done}. This function is
* invoked by the function activation_update and effectively
```

```
* synchronizes the completion of the function execution.
*
*****************************************************************/

THREADED
compute_change_of_state(float change, SPTR done)
{
 change_of_state += change;
 RSYNC(done);
 END_FUNCTION();
}



/*****************************************************************
*
* THREAD_0 allocates the memory necessary for the a_old array.
* THREAD_0 is also responsible for invoking the I_READ_F function
* for each element of the I-structure array.
*
* Because the sync slot is initialized with {\tt num\_neurons}, {\tt
* THREAD\_1} is not spawned until all the read operations are
* serviced. {\tt THREAD\_1} computes the new activation for the
* neuron {\tt NODE\_ID}, invokes the {\tt I\_WRITE\_F} function to
* write this new activation value to the {\tt new} I-structure,
* computes the square of the amount of change in the activation
* value and reports this change to node 0 invoking the function {\tt
* compute\_change\_of\_state()}. This later function synchronizes the
```

```
* sync slot {\tt done} to signal that the activation update is
* complete.
*
*****************************************************************/

THREADED
activation_update(int neuron_id, float *new, float *old, SPTR done)
{
    SLOT            SYNC_SLOTS[2];
    float           *a_old;
    float           activation;
    float           change;
    int             i;

    INIT_SYNC(0, NET_SIZE, NET_SIZE, 1);

#ifdef DEBUG
    fprintf(stderr, "Activation Update for neuron#%d...\n",
            neuron_id);
#endif
    a_old = (float *) malloc(NET_SIZE*sizeof(float));
    for(i=0 ; i<NET_SIZE ; i++) {
      /*
#ifdef CACHE
      INVOKE(NODE_ID, SC_I_READ, OWNER(i), old, POSITION(i), F,
     TO_GLOBAL(&a_old[i]), SLOT_ADR(0));
#else
      INVOKE(NODE_ID, NOW_I_READ, OWNER(i), old, POSITION(i), F,
```

```
     TO_GLOBAL(&a_old[i]), SLOT_ADR(0));
#endif
*/

      INVOKE(NODE_ID, NOW_GET_RSYNC,
     MAKE_GPTR((float *)old+POSITION(i), OWNER(i)),
     TO_GLOBAL(&a_old[i]), F, SLOT_ADR(0));

      }
    END_THREAD();

 THREAD_1:
#ifdef DEBUG
      fprintf(stderr,"Node %d - activation update: Spawned THREAD_1.
              \n", NODE_ID);
#endif
    activation = 0;
    for(i=0 ; i<NET_SIZE ; i++)
       activation += synapse[POSITION(neuron_id) * NET_SIZE+i]
                     * a_old[i];
#ifdef DEBUG
    fprintf(stderr, "Node %d updates activation=%f\n", NODE_ID,
            activation);
#endif
    activation = (activation > 0.0) ? +1.0 : -1.0;
#ifdef DEBUG
    fprintf(stderr, "activation turned to =%f\n", NODE_ID,
            activation);
```

```
#endif
    /*
    INVOKE(OWNER(neuron_id), I_WRITE_F, new, POSITION(neuron_id),
   activation);
   */
    INIT_SYNC(1, 1, 1, 2);
    DATA_RSYNC_F(activation, MAKE_GPTR((float *)new +
                 POSITION(neuron_id),OWNER(neuron_id)), SLOT_ADR(1));

#ifdef DEBUG
    fprintf(stderr, "Old activation=%f\n", a_old[neuron_id]);
#endif

    change = (activation - a_old[neuron_id]);
    change = change*change;
#ifdef DEBUG
    fprintf(stderr,"change=%f\n", change);
#endif
    INVOKE(0, compute_change_of_state, change, done);
    free(a_old);
    END_THREAD();
 THREAD_2:
    END_FUNCTION();
}

THREADED
InitGlobal(int dim, int ni_delay, SPTR done)
{
```

```
  int i,j;

  NET_SIZE = dim;
  NI_DELAY = ni_delay;

  /* Allocate and initialize synapses array */
  synapse = (float *)malloc(NET_SIZE * NET_SIZE / NUM_NODES *
            sizeof(float));

  for(i=0; i< (NET_SIZE/NUM_NODES); i++) {
    for(j=0; j<NET_SIZE; j++) {
      synapse[i*NET_SIZE + j] = 0.01*(IMAP(NODE_ID,i)+1)*j;
    }
  }
#ifdef DEBUG
  printf("\nNI_DELAY = %d\n",ni_delay);
#endif DEBUG

  RSYNC(done);

  END_FUNCTION();
}

THREADED
LocalAllocate(int num, SPTR done)
{
  /*
```

```
  INVOKE(NODE_ID, I_ALLOCATE, num/NUM_NODES, TO_GLOBAL(&i_old),
         SLOT_ADR(0));
  INVOKE(NODE_ID, I_ALLOCATE, num/NUM_NODES, TO_GLOBAL(&i_new),
         SLOT_ADR(0));
  */
  i_old = Array1;
  i_new = Array2;
  RSYNC(done);
  END_FUNCTION();
}

THREADED
RESET_I_NEW(SPTR done)
{
  float *temp;

  temp = i_old;
  i_old = i_new;
  i_new = temp;
  RSYNC(done);
  END_FUNCTION();
}


THREADED
MAIN(int argc, char** argv)
{
    SLOT            SYNC_SLOTS[5];
```

```
  /*
  void *GLOBAL i_old;
  void *GLOBAL i_new;
  void *GLOBAL temp;
  */
  float         *final;
  int           i, par_no;
  int           flip;
  int           num_neurons;
  int           num_iter;
  unsigned long t1,t2,dt1,dt2,delay_time;

NI_DELAY = 0;
NET_LATENCY = 0;
NET_SIZE = 16;

if(argc > 1) {
  par_no = 0;
  while(par_no < argc) {
    if(!strcmp(argv[par_no], "-ni")) {
      sscanf(argv[par_no+1], "%d", &NI_DELAY);
      par_no = par_no+2;
    } else if(!strcmp(argv[par_no], "-nl")) {
      sscanf(argv[par_no+1], "%d", &NET_LATENCY);
      par_no = par_no +2;
    } else if(!strcmp(argv[par_no], "-d")) {
      sscanf(argv[par_no+1], "%d", &NET_SIZE);
      par_no = par_no +2;
```

```
      } else {
        par_no++;
      }
    }
  }

    num_neurons = NET_SIZE;

    INIT_SYNC(0,NUM_NODES,NUM_NODES,1);
    INIT_SYNC(1,NUM_NODES,NUM_NODES,2);
    INIT_SYNC(2,num_neurons,NUM_NODES,3);
    INIT_SYNC(3,num_neurons,num_neurons,4);
    INIT_SYNC(4,num_neurons,num_neurons,5);

    final = (float *)malloc(num_neurons*sizeof(float));

    for(i=0; i<NUM_NODES; i++)
      INVOKE(i, InitGlobal, NET_SIZE, NI_DELAY, SLOT_ADR(0));

    END_THREAD();

 THREAD_1:
#ifdef DEBUG
    fprintf(stderr,"MAIN: Allocating i_old and i_new\n");
#endif
    /* Allocates two I-Structures i_old and i_new on each nodes */
    for(i=0; i<NUM_NODES; i++)
      INVOKE(i, LocalAllocate, num_neurons, SLOT_ADR(1));
```

```
    END_THREAD();


 THREAD_2:  /* synchronized by I_ALLOCATEs of i_old and i_new */
#ifdef DEBUG
    fprintf(stderr,"MAIN: Initializing i_old.\n");
#endif


    flip = -1.0;
    for(i=0 ; i<num_neurons ; i++)
      {
flip = -1.0*flip;
/*
INVOKE(OWNER(i), I_WRITE_F, i_old, POSITION(i),
       flip*0.01*(float)(i+1));
       */
DATA_RSYNC_F(flip*0.01*(float)(i+1),
     MAKE_GPTR((float *)i_old+POSITION(i),
                     OWNER(i)), SLOT_ADR(2));

      }


    num_iter=0;
    dt1 = ct_read();
    delay(NI_DELAY);
    dt2 = ct_read();
    delay_time = (dt2-dt1)/25;
```

```
    END_THREAD();

 THREAD_3:
#ifdef DEBUG
    fprintf(stderr,"MAIN: Activation Update\n");
#endif
    if(num_iter == 0) t1=ct_read();
    num_iter++;
    change_of_state = 0.0;
    for(i=0 ; i<num_neurons ; i++)
      INVOKE(OWNER(i), activation_update, i, i_new, i_old,
             SLOT_ADR(3));
    END_THREAD();

 THREAD_4:
#ifdef DEBUG
    fprintf(stderr,"MAIN: Criterium check.\n");
#endif
    /*
    temp = i_old;
    i_old = i_new;
    i_new = temp;
    */
#ifdef DEBUG
    fprintf(stderr, "      => change_of_state = %f\n",
            change_of_state);
#endif
```

```
    if(change_of_state > STOPPING_CRITERIUM) {
      for(i=0; i<NUM_NODES; i++)
INVOKE(i, RESET_I_NEW, SLOT_ADR(2));
    } else {
      t2 = ct_read();
      /*
      for(i=0; i<NUM_NODES; i++)
INVOKE(i, I_DELETE, i_old);
*/
      for(i=0 ; i<num_neurons ; i++)
/*
  INVOKE(OWNER(i), I_READ_F, i_new, POSITION(i),
 TO_GLOBAL(&final[i]), SLOT_ADR(4));
 */
INVOKE(NODE_ID, NOW_GET_RSYNC, MAKE_GPTR(i_new+POSITION(i),
                OWNER(i)), TO_GLOBAL(&final[i]), F, SLOT_ADR(4));

    }

    END_THREAD();


 THREAD_5:
#ifdef DEBUG
    fprintf(stderr,"MAIN: Finishing.\n");
#endif

    free(final);
```

```
    printf("Number of iteration=%d\n",num_iter);
    printf("Execution time = %dus\n", (t2-t1)/25);
    CALL(Gather_Cache_Stat, -1, NET_SIZE, delay_time, (t2-t1)/25);
    RETURN();
}
```

B.2 Makefile

```
CC = etcc
CFLAGS = -O4
TARGET = -target manna-spn
INCLUDE = ~wenyenl/lib/i-struct
LIB = /m/capslguests/wenyenl/lib/i-struct/i-struct.o
          /m/capslguests/wenyenl/lib/i-struct/issc.o

all: hopfield_is hopfield_issc

hopfield_is: hopfield1.c
$(CC) $(CFLAGS) -I$(INCLUDE) $(TARGET) $(LIB)
                -o hopfield_is hopfield1.c

hopfield_issc: hopfield1.c
$(CC) $(CFLAGS) -I$(INCLUDE) $(TARGET) $(LIB)
                -DCACHE -o hopfield_issc hopfield1.c

clean:
-rm -f *.o core hopfield_is hopfield_issc
```